

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



M a s t e r i n g

精通区块链 开发技术

(美) 伊姆兰·巴希尔 | 著

王烈征 | 译

B l o c k c h a i n

清华大学出版社





精通区块链开发技术

(美) 伊姆兰·巴希尔 著

王烈征 译

清华大学出版社

北 京



内 容 简 介

本书详细阐述了与区块链开发相关的基本解决方案,主要包括区块链、去中心化、密码学和基本技术、比特币、替代币、智能合约、以太坊、超级账本等内容。此外,本书还提供了相应的示例、代码,以帮助读者进一步理解相关方案的实现过程。

本书适合作为高等院校计算机及相关专业的教材和教学参考书,也可作为相关开发人员的自学教材和参考手册。

Copyright © Packt Publishing 2017. First published in the English language under the title
Mastering Blockchain.

Simplified Chinese-language edition © 2018 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Packt Publishing 授权清华大学出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2017-7182

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

精通区块链开发技术/(美)伊姆兰·巴希尔(Imran Bashir)著;王烈征译. —北京:清华大学出版社, 2018

书名原文: Mastering Blockchain

ISBN 978-7-302-49983-1

I. ①精… II. ①伊… ②王… III. ①电子商务-支付方式-研究 IV. ①F713.361.3

中国版本图书馆CIP数据核字(2018)第069053号

责任编辑: 贾小红

封面设计: 刘超

版式设计: 魏远

责任校对: 马子杰

责任印制: 丛怀宇

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦A座 邮 编: 100084

社总机: 010-62770175 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者: 三河市君旺印务有限公司

经 销: 全国新华书店

开 本: 185mm×230mm 印 张: 25.5 字 数: 525千字

版 次: 2018年6月第1版 印 次: 2018年6月第1次印刷

印 数: 1~3000

定 价: 129.00元

产品编号: 077465-01



译者序

在阅读本书之前，相信读者已对区块链及其巨大的发展潜力有所耳闻。

2008年，随着比特币的出现，当今世界步入一个全新的概念中，并很有可能引发全社会的变革，同时对各行各业产生深远的影响，其中包括（但不仅限于）金融业、政府部门以及媒体。一些人把区块链描述为一场革命，而另一种思想学派则认为，这将是一种进化，且需要许多年才能从区块链获得实际利益。这在某种程度上是正确的，但我认为变革已经开始了；世界上许多大型机构已经开始用区块链技术来证明这一概念，因为其颠覆性潜力已经得到充分的承认。然而，一些组织仍处于初步探索阶段，但随着技术的成熟，预计将会更快地取得进展。

本书详细阐述了与区块链开发相关的基本解决方案，主要包括区块链、去中心化、密码学和基本技术、比特币、智能合约、以太坊、超级账本等内容。此外，本书还提供了相应的示例、代码，以帮助读者进一步理解相关方案的实现过程。

在本书的翻译过程中，除王烈征之外，黄立臣、周建娟、李秋霞、程晓磊、于鑫睿、张博、刘伟、张骞、李垚、张颖、张弢、刘君、李强、沈旻、李伟、李娇娇、翟露洋、刘洋、蔡辉、王福会、杨崇珉、刘璋、刘晓雪、张华臻等人也参与了本书的翻译工作，在此一并表示感谢。

限于译者的水平，译文中难免有错误和不妥之处，恳请广大读者批评指正。

译者



前言

本书将全面介绍区块链技术的理论和实践，涵盖了充分理解区块链技术的全部内容。在阅读完本书后，读者将能够深入了解区块链技术的内部工作原理，并具备开发区块链应用程序的能力。本书包含了与区块链技术相关的所有主题，涉及密码学、加密货币、比特币、以太坊，以及用于区块链开发的各种平台和工具。

在阅读本书之前，建议读者具备一定的计算机科学知识和基本的编程经验，但若无经验也并不妨碍读者的学习进程，相关背景知识在书中均有所介绍。

本书内容

第1章介绍了基于区块链技术的分布式计算的基本概念，包括区块链的历史、定义、特性、类型和优点，以及区块链技术的核心内容——共识机制。

第2章介绍了去中心化的概念及其与区块链技术的关系。除此之外，还讨论了相关方法和平台，进而可对处理过程或系统执行去中心化操作。

第3章介绍了密码学的理论基础，这也是全面了解区块链技术的必要条件，其中包括公钥和私钥加密等概念及其应用实例。最后，本章还对金融市场进行了简要介绍，在金融领域，许多有趣的用例均可实现于区块链技术中。

第4章主要讨论比特币，这也是第一个最大的区块链。本章详细介绍了与比特币加密有关的技术概念。

第5章讨论了比特币出现之后的其他替代加密货币，包括各种代币示例、属性以及开发和实现方式。

第6章深入讨论了智能合约，介绍了智能合约的历史、定义、李嘉图合约、Oracle 定义，以及智能合约的理论知识。

第7章详细介绍了以太坊区块链的设计和架构，包括与以太坊区块链相关的各种技术概念，并深入分析了该平台的基本原理、特性和组件。

第8章考查了一个详细的示例，并使用以太坊区块链开发去中心化的应用程序和智能合约。除此之外，本章还讨论了 Solidity 语言和各種相关工具。



第 9 章介绍了 Linux 的超级账本项目，其中包含了不同的区块链项目。

第 10 章介绍了可替代的区块链解决方案和平台，同时还阐述了可替代区块链的技术细节和特性。

第 11 章探讨了区块链技术在其他领域的应用，包括物联网、政府部门、媒体和金融等行业。

第 12 章考查了区块链技术所面临的一些挑战性问题及其解决方案。

第 13 章讨论与区块链技术的现状、项目和研究工作相关的信息。此外，还提出了一些针对区块链技术现状的预测观点。

准备工作

本书中的所有示例都是在 Ubuntu 16.04.1 LTS (Xenial) 上进行开发的。因此，建议读者使用 Ubuntu 系统。但这并不意味着排除了其他操作系统，例如 Windows 或 Linux，但相关示例，尤其是与安装相关的操作步骤，可能需要进行相应的更改。

与密码学相关的示例使用 OpenSSL 1.0.2g 1 Mar 2016 命令行工具进行开发。

以太坊的 Solidity 示例则通过 Browser Solidity 予以实现，读者可访问 <https://ethereum.github.io/browser-solidity/> 获取在线资源。

以太坊的 homestead 版本可用于开发相关应用示例，在本书编写时，这也是以太坊的最新版本，读者可访问 <https://www.ethereum.org/> 进行下载。

与物联网相关的示例可通过 Raspberry Pi 工具包进行开发。特别地，可采用 Raspberry Pi 3 Model B V 1.2 创建物联网硬件示例。Node.js V7.2.1 和 npm V3.10.10 则用于下载相关的数据包，并针对物联网示例运行 Node.js 服务器。

Truffle 框架用于智能合约的部署操作，读者可访问 <http://truffleframework.com/> 进行下载。

适用读者

本书适用于希望深入了解区块链技术的读者。另外，本书也可用作区块链应用程序开发人员的参考工具书，同时也可作为区块链技术和加密货币相关课程的教学参考书，以及各种考试和认证的学习资源。



本书约定

代码块通过下列方式设置：

```
function difference(uint x) returns (uint y)
{
    z=x-5;
    y=z;
}
```

代码中的重点内容则采用黑体表示：

```
function difference(uint x) returns (uint y)
{
    z=x-5;
    y=z;
}
```

命令行输入或输出如下所示：

```
$ geth --datadir .ethereum/PrivateNet/ --networkid 786 --rpc --
rpccorsdomain 'http://192.168.0.17:9900'
```



图标表示较为重要的说明事项。



图标则表示提示信息和操作技巧。

读者反馈和客户支持

欢迎读者对本书提出建议或意见，以帮助我们进一步解读者的阅读喜好。反馈意见对于我们来说十分重要，以便改进我们日后的工作。对此，读者可向 feedback@packtpub.com 发送邮件，并以书名作为邮件标题。若读者针对某项技术具有专家级的见解，抑或计划撰写书籍或完善某部著作的出版工作，则可访问 www.packtpub.com/authors。



我们将为每一位本书读者竭诚服务。

资源下载

读者可访问 <http://www.packtpub.com> 并通过个人账户下载示例代码文件。另外，可访问 <http://www.packtpub.com/support>，注册成功后，我们将以电子邮件的方式将相关文件发与读者。

读者可根据下列步骤下载代码文件：

- ☐ 通过个人电子邮件地址和密码登录并注册我们的网站。
- ☐ 选择 SUPPORT 选项卡。
- ☐ 单击 Code Downloads & Errata。
- ☐ 在 Search 文本框中输入书名。
- ☐ 选择本书对应的代码文件。
- ☐ 从下拉菜单中选择本书的购买方式。
- ☐ 单击 Code Download。

当文件下载完毕后，确保使用下列最新版本软件解压文件夹：

- ☐ Windows 系统下的 WinRAR/7-Zip。
- ☐ Mac 系统下的 Zipeg/iZip/UnRarX。
- ☐ Linux 系统下的 7-Zip/PeaZip。

另外，读者还可访问 GitHub 获取本书的代码包，对应网址为 <https://github.com/PacktPublishing/Mastering-Blockchain>。此外，读者还可访问 <https://github.com/PacktPublishing/> 以了解丰富的代码和视频资源。

读者可访问 https://www.packtpub.com/sites/default/files/downloads/MasteringBlockchain_ColorImages.pdf 下载本书的彩色图像，以方便读者对比某些输出结果。

勘误表

尽管我们在最大程度上做到尽善尽美，但错误依然在所难免。如果读者发现谬误之处，无论是文字错误抑或是代码错误，还望不吝赐教。对于其他读者以及本书的再版工作，这将有十分重要的意义。对此，读者可访问 <http://www.packtpub.com/submit-errata>，选取对



应书籍，单击 Errata Submission Form 超链接，并输入相关问题的详细内容。经确认后，填写内容将被提交至网站，或添加至现有勘误表中（位于该书籍的 Errata 部分）。

另外，读者还可访问 <http://www.packtpub.com/books/content/support> 查看之前的勘误表。在搜索框中输入书名后，所需信息将显示于 Errata 项中。

版权须知

一直以来，互联网上的版权问题从未间断，Packt 出版社对此类问题异常重视。若读者在互联网上发现本书任意形式的副本，请告知网络地址或网站名称，我们将对此予以处理。

关于盗版问题，读者可发送邮件至 copyright@packtpub.com。

对于作者的爱护，我们表示衷心的感谢，并将于日后向读者呈现更为精彩的作品。

问题解答

若读者对本书有任何疑问，均可发送邮件至 questions@packtpub.com，我们将竭诚为您服务。

目 录

第 1 章 区块链.....	1
1.1 分布式系统.....	2
1.1.1 CAP 定理.....	3
1.1.2 拜占庭将军问题.....	4
1.1.3 一致性.....	4
1.2 区块链发展史.....	5
1.2.1 电子现金.....	6
1.2.2 电子现金的概念.....	6
1.3 区块链简介.....	8
1.3.1 区块链技术的各种定义.....	9
1.3.2 区块链中的一般元素.....	10
1.3.3 区块链特性.....	11
1.3.4 区块链技术应用.....	13
1.3.5 区块链发展层次.....	13
1.4 区块链类型.....	14
1.4.1 公有区块链.....	14
1.4.2 私有区块链.....	14
1.4.3 半私有区块链.....	15
1.4.4 侧链技术.....	15
1.4.5 许可账本.....	15
1.4.6 分布式账本.....	15
1.4.7 共享账本.....	15
1.4.8 全私有和专有区块链.....	15
1.4.9 标记化区块链.....	16
1.4.10 无代币区块链.....	16
1.4.11 区块链中的共识.....	16
1.5 CAP 定理和区块链.....	18

1.6	区块链的优点和局限性	18
1.7	区块链技术的限制和挑战	19
1.8	本章小结	20
第 2 章	去中心化	21
2.1	基于区块链的去中心化	21
2.2	去中心化方法	23
2.2.1	非中介化	23
2.2.2	竞争	23
2.3	去中心化流程	24
2.4	区块链和完整的生态圈去中心化操作	25
2.4.1	存储	25
2.4.2	通信	26
2.4.3	计算	27
2.5	智能合约	28
2.6	去中心化组织	28
2.7	去中心化自治组织	29
2.8	去中心化自治企业	29
2.9	去中心化自治社会	30
2.10	去中心化应用程序	30
2.10.1	去中心化应用程序的需求条件	30
2.10.2	DAPP 操作	31
2.11	去中心化平台	31
2.12	本章小结	32
第 3 章	密码学和基本技术	33
3.1	简介	33
3.1.1	数学知识	33
3.1.2	密码学	35
3.1.3	保密性	35
3.1.4	完整性	35
3.1.5	认证	35
3.1.6	不可否认性	36

3.1.7 问责制	36
3.2 密码原语	37
3.2.1 对称加密	38
3.2.2 块密码	39
3.2.3 数据加密标准	42
3.2.4 高级加密标准 (AES)	42
3.3 非对称加密	45
3.3.1 整数分解	47
3.3.2 离散对数	47
3.3.3 椭圆曲线	47
3.4 公钥和私钥	48
3.4.1 RSA	48
3.4.2 离散对数问题	54
3.4.3 密码原语	62
3.4.4 哈希函数	62
3.4.5 椭圆曲线数字签名算法 (ECDSA)	71
3.5 金融市场和交易	76
3.5.1 交易	77
3.5.2 交易所	77
3.5.3 交易的生命周期	78
3.5.4 订单预期者	79
3.5.5 市场操控	79
3.6 本章小结	79
第 4 章 比特币	81
4.1 比特币概述	82
4.1.1 比特币的概念	83
4.1.2 密钥和地址	83
4.1.3 比特币中的公钥	84
4.1.4 比特币中的私钥	84
4.1.5 比特币货币单位	85
4.1.6 Base58Check 编码	85

4.1.7 虚地址	86
4.2 交易/事务	87
4.2.1 交易的生命周期	87
4.2.2 交易的结构	87
4.2.3 交易类型	90
4.3 区块链	94
4.3.1 区块链结构	94
4.3.2 区块头结构	94
4.3.3 创始区块	96
4.3.4 比特币网络	103
4.3.5 钱包	109
4.4 比特币支付	112
4.4.1 比特币投资和比特币交易	113
4.4.2 比特币安装	114
4.4.3 比特币编程和命令行接口	120
4.4.4 比特币改进协议 (BIP)	120
4.5 本章小结	121
第 5 章 替代币	123
5.1 理论基础	125
5.1.1 工作量证明的替代方案	125
5.1.2 难度调整和目标重定位算法	128
5.2 比特币中的限制条件	130
5.2.1 隐私和匿名性	130
5.2.2 比特币上的扩展协议	131
5.2.3 替代币的开发	133
5.3 域名币	135
5.4 莱特币	140
5.5 素数币	142
5.5.1 素数币交易	143
5.5.2 挖掘规则	144
5.6 Zcash	145

5.6.1	Zcash 交易	146
5.6.2	挖掘规则	147
5.6.3	GPU 挖掘	150
5.7	本章小结	152
第 6 章	智能合约	153
6.1	发展历史	153
6.2	定义	153
6.3	李嘉图合约	155
6.3.1	智能合约模板	158
6.3.2	Oracle	159
6.3.3	Smart Oracle	160
6.3.4	在区块链上发布智能合约	160
6.3.5	DAO	161
6.4	本章小结	161
第 7 章	以太坊	163
7.1	简介	163
7.1.1	以太坊客户端和发布	163
7.1.2	以太坊栈	164
7.2	以太坊区块链	164
7.2.1	货币 (ETH 和 ETC)	165
7.2.2	分叉	165
7.2.3	gas	166
7.2.4	共识机制	166
7.2.5	世界状态	167
7.2.6	交易	168
7.2.7	合约生成型交易	170
7.2.8	消息调用型交易	171
7.3	以太坊区块链中的元素	172
7.3.1	以太坊虚拟机	172
7.3.2	执行环境	173
7.3.3	操作码及其含义	176

7.4	预编译合同	182
7.4.1	椭圆曲线公钥恢复函数	182
7.4.2	SHA256 位哈希函数	182
7.4.3	RIPEMD160 位哈希函数	182
7.4.4	恒等函数	182
7.5	账户	183
7.6	区块	183
7.6.1	区块头	184
7.6.2	创始区块	185
7.6.3	交易收据	186
7.6.4	交易验证和执行	186
7.6.5	区块验证机制	187
7.7	Ether	189
7.7.1	gas	189
7.7.2	费用标准	190
7.8	消息	190
7.9	挖掘	191
7.9.1	Ethash	192
7.9.2	CPU 挖掘	192
7.9.3	GPU 挖掘	193
7.9.4	挖掘设备	194
7.10	客户端和矿工	196
7.11	贸易与投资	204
7.12	黄皮书	205
7.13	以太坊网络	206
7.13.1	MainNet	206
7.13.2	TestNet	206
7.13.3	专用网络	206
7.14	所支持的协议	207
7.15	以太坊应用程序	208
7.16	可扩展性和安全问题	208
7.17	本章小结	208

第 8 章 以太坊开发	211
8.1 配置开发环境	211
8.1.1 TestNet (Ropsten)	211
8.1.2 配置 PrivateNet	212
8.1.3 启动私有网络	214
8.1.4 在 PrivateNet 上运行 Mist	218
8.1.5 利用 Mist 部署合约	219
8.2 开发工具和客户端	223
8.2.1 开发语言	224
8.2.2 编译器	224
8.2.3 工具和库	228
8.2.4 EthereumJS	230
8.2.5 合约的开发和部署	231
8.3 Solidity 语言	231
8.3.1 值类型	232
8.3.2 字面值	233
8.3.3 枚举值	234
8.3.4 函数类型	234
8.3.5 引用类型	234
8.3.6 映射	235
8.3.7 全局变量	236
8.3.8 控制结构	236
8.4 引入 Web3	241
8.4.1 POST 请求	247
8.4.2 HTML 和 JavaScript 前端	248
8.4.3 开发框架	255
8.5 本章小结	281
第 9 章 超级账本	283
9.1 项目	283
9.1.1 Fabric	283
9.1.2 Sawtooth lake	283

9.1.3	Iroha.....	284
9.1.4	Blockchain explorer.....	284
9.1.5	Fabric 链式工具	284
9.1.6	Fabric SDK Py.....	284
9.1.7	Corda	285
9.2	超级账本协议	285
9.2.1	参考架构	285
9.2.2	需求条件	286
9.2.3	隐私和保密性	286
9.2.4	身份	287
9.2.5	可审核性	287
9.2.6	互操作性	287
9.2.7	可移植性	287
9.3	Fabric	287
9.4	Hyperledger Fabric	288
9.4.1	Fabric 体系结构	288
9.4.2	Fabric 组件	291
9.5	Sawtooth lake	293
9.5.1	PoET	293
9.5.2	交易族	293
9.5.3	Sawtooth 中的共识机制	295
9.5.4	开发环境	295
9.6	Corda	298
9.6.1	体系结构	299
9.6.2	组件	300
9.6.3	开发环境	302
9.7	本章小结	303
第 10 章	替代区块链方案	305
10.1	区块链	305
10.2	平台	318
10.2.1	BlockApps	318

10.2.2 Eris.....	324
10.3 本章小结	326
第 11 章 货币之外的区块链技术	327
11.1 物联网	327
11.2 政府机构	344
11.2.1 边境管理	344
11.2.2 选票机制	346
11.2.3 身份证	346
11.2.4 其他领域	347
11.3 保健事业	347
11.4 金融行业	348
11.4.1 保险行业	348
11.4.2 交易后的结算	349
11.4.3 防范金融犯罪	349
11.5 媒体行业	350
11.6 本章小结	350
第 12 章 可扩展性和其他挑战.....	351
12.1 可扩展性	351
12.1.1 增加区块链尺寸	352
12.1.2 减少区块间隔时间	352
12.1.3 可逆的 Bloom 查找表	353
12.1.4 分片技术	353
12.1.5 状态通道	353
12.1.6 私有区块链	354
12.1.7 权益证明	354
12.1.8 侧链	354
12.1.9 子链	354
12.1.10 树形链	354
12.2 隐私性	356
12.2.1 不可区分性混淆技术	356
12.2.2 同态加密	356

12.2.3	零知识证明	356
12.2.4	状态通道	357
12.2.5	安全的多方计算	357
12.2.6	通过硬件提供保密性	357
12.2.7	Coinjoin	357
12.2.8	机密交易	358
12.2.9	MimbleWimble	358
12.3	安全性	358
12.3.1	智能合约安全性	359
12.3.2	Why3 形式验证	360
12.3.3	Oyente 工具	361
12.4	本章小结	362
第 13 章	发展现状和未来趋势	365
13.1	新趋势	365
13.1.1	基于应用程序的区块链 (ASBC)	365
13.1.2	企业级区块链	365
13.1.3	私有区块链	366
13.1.4	初创公司	366
13.1.5	浓厚的研究兴趣	366
13.1.6	标准化	367
13.1.7	改进措施	367
13.1.8	具体实现	368
13.1.9	企业联合体	368
13.1.10	解决方法	368
13.1.11	技术融合	368
13.1.12	教育发展状况	368
13.1.13	就业前景	369
13.1.14	密码经济学	369
13.1.15	密码学研究	369
13.1.16	新的编程语言	369
13.1.17	硬件研究和开发	370

13.1.18	形式方法和以及安全研究	370
13.1.19	区块链的替代方案	370
13.1.20	互操作性	371
13.1.21	区块链服务	371
13.1.22	减少耗电量	371
13.2	改进协议	371
13.2.1	BIP	372
13.2.2	EIP	373
13.3	其他挑战性问题	374
13.4	负面影响	375
13.5	区块链研究	376
13.5.1	智能合约	376
13.5.2	中心化问题	376
13.5.3	加密功能的局限性	376
13.5.4	共识算法	376
13.5.5	可扩展性	377
13.5.6	代码混淆	377
13.6	重要项目实例	377
13.6.1	以太坊上的 Zcash	377
13.6.2	CollCo	377
13.6.3	Cello	378
13.6.4	Qtum	378
13.6.5	Bitcoin-NG	378
13.6.6	Solidus	378
13.6.7	Hawk	378
13.6.8	Town-Crier	378
13.6.9	SETLCoin	379
13.6.10	TEEChan	379
13.6.11	Falcon	379
13.6.12	Bletchley	379
13.6.13	Casper	380
13.6.14	Metropolis	380

13.7 其他工具	380
13.7.1 Microsoft Visual Studio 的 Solidity 扩展	380
13.7.2 MetaMask	380
13.7.3 Stratis	381
13.7.4 Embark	381
13.7.5 DAPPLE	381
13.7.6 Meteor	381
13.7.7 uPort	381
13.7.8 INFURA	382
13.8 与其他行业的结合	382
13.9 未来发展	383
13.10 本章小结	384

第1章 区块链

在阅读本书之前，相信读者已对区块链及其巨大的发展潜力有所耳闻。

2008 年，随着比特币的出现，为当今世界带来一个全新的概念，并很有可能引发全社会的变革，同时对各行各业产生深远的影响，其中包括（但不限于）金融业、政府部门以及媒体。一些人把区块链描述为一场革命，而另一种思想学派则认为，这将是一种进化，且需要许多年才能从区块链获得实际利益。这在某种程度上是正确的，但我认为革命已经开始了；世界上许多大型机构已经开始用区块链技术来证明这一概念，因为其颠覆性潜力已经得到了充分的承认。然而，一些组织仍处于初步探索阶段，但随着技术的成熟，预计将会更快地取得进展。区块链技术对当前的技术也有影响，并具有在基本水平上改变它们的能力。

据 Gartner 的技术成熟度曲线图显示，区块链技术目前处于膨胀预期的顶峰（截至 2016 年 7 月，如图 1.1 所示），预计将在 5~10 年内为主流应用。

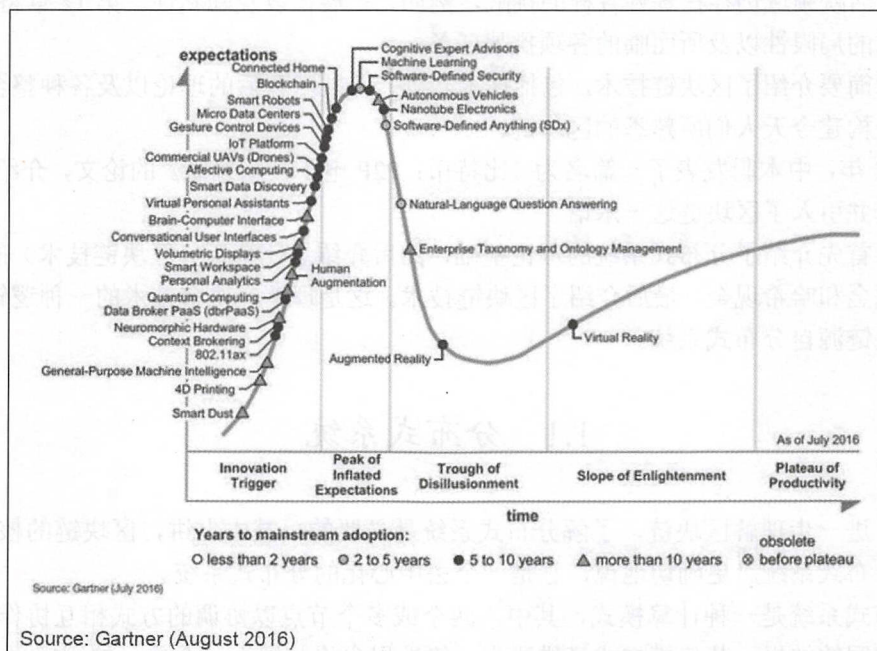


图 1.1 新兴技术的技术成熟度曲线图

从密码币（cryptocurrency）角度来看，区块链曾一度被认为是一类奇特的货币，且未受到足够的重视，但在过去的几年里，人们对区块链技术的兴趣大增；当今，一些公司和组织都在研究这一技术，花费了数百万美元尝试应用这项技术，并对其展开各项试验。图 1.2 显示了过去这几年中，Google 公司的区块链技术趋势分析。

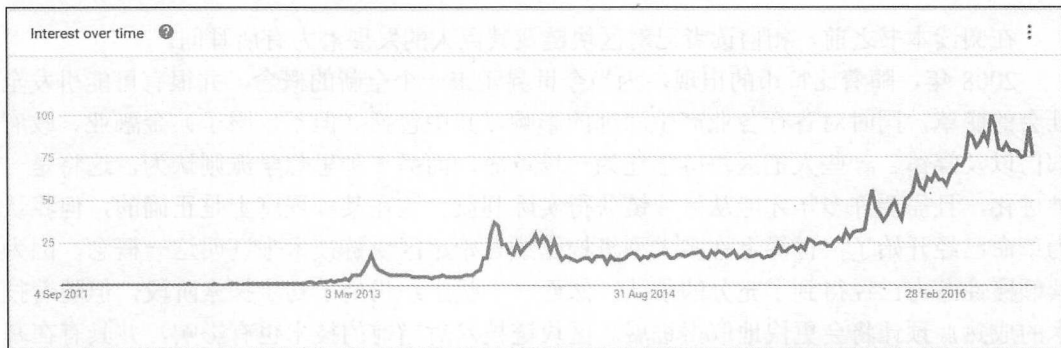


图 1.2 Google 区块链技术趋势

区块链技术的优势包括去中心化信任机制、成本节约、透明度以及高效性。尽管如此，这一活跃领域仍存在各种各样的挑战，例如，扩展性以及隐私性。第 12 章将讨论区块链技术的局限性以及所面临的各项挑战任务。

本章简要介绍了区块链技术，包括技术基础、该技术背后的理论以及各种整合技术，进而用于构建今天人们所熟悉的区块链。

2008 年，中本聪发表了一篇名为《比特币：P2P 电子货币系统》的论文，介绍了 P2P 电子货币并引入了区块链这一术语。

本章首先介绍了分布式系统的理论基础，然后介绍了比特币（区块链技术）的前身，如电子现金和哈希现金，最后介绍了区块链技术。这是理解区块链技术的一种逻辑方式，因为区块链源自分布式系统。

1.1 分布式系统

为了进一步理解区块链，了解分布式系统是必要的。基本上讲，区块链的核心内容是一个分布式系统。更确切地说，它是一个去中心化的分布式系统。

分布式系统是一种计算模式，其中，两个或多个节点以协调的方式相互协作，以实现一个共同的结果；其建模方式可描述为：终端用户将其视为一个单一的逻辑平台。

一个节点可以被定义为分布式系统中的个体成员，所有节点都能够相互发送和接收

消息。其中，节点可以是可信节点，或者是具有缺陷的节点或者是恶意的节点，并且包含自身的内存和处理器。一个可以展示任意行为的节点也被称为拜占庭节点。这种任意行为可能包含了某种恶意操作，并对网络的运行带来负面影响。通常，网络上某个节点的任何意外行为都可以归类为拜占庭节点。此类节点通常包含了意外或恶意的行为，如图 1.3 所示。

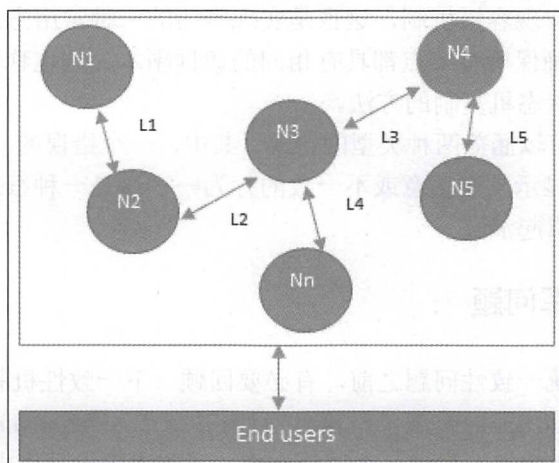


图 1.3 分布式系统设计。其中，N4 表示为拜占庭节点；L2 表示为断裂的或缓慢的网络链接

分布式系统设计中的主要挑战是节点与容错之间的协调。即使某些节点出现故障或网络链接中断，分布式系统也应该容忍这一类情况，并应该继续完美地工作，以达到预期的结果。多年来，这一直是一个活跃的研究领域，并提出了一些算法和机制来克服此类问题。

分布式系统设计具有较大的挑战性，以至于制定了一个称为 CAP 的定理，该定理指出：分布式系统不能同时拥有所期望的属性。在 1.1.1 节中，将提供关于 CAP 定理的基本介绍。

1.1.1 CAP 定理

CAP 定理也被称为布鲁尔定理，最初是由埃里克·布鲁尔在 1998 年提出的猜想；2002 年，赛斯·吉尔伯特和南希·林奇将其证明为一个定理。

CAP 定理指出，任何分布式系统都不能同时具有一致性、可用性和分区容错性。

- ❑ 一致性是一种属性，它确保分布式系统中的所有节点都有一个最新的数据副本。
- ❑ 可用性意味着系统在使用过程中具有可访问性，接收传入的请求并在必要时对

数据做出应有的响应。

□ 分区容错性确保如果一组节点出现故障，分布式系统仍然能够正确地运行。

已经证明，分布式系统不能同时具备上述 3 个属性。这听起来似乎有些奇怪，因为区块链能够实现所有这些属性，抑或这只是一个噱头？关于这一类内容，将在后续章节中对区块链上下文中的 CAP 定理加以解释。

可采用复制方式实现容错机制，这也是实现容错的一种常用方法。一致性是通过一致性算法实现的，以确保所有节点都具有相同的数据副本，这也称为状态机复制。区块链基本上是一种实现状态机复制的方法。

一般来说，节点可以涵盖两种类型的错误。其中，一个错误的节点只是简单地崩溃；而有缺陷的节点可以显示具有恶意或不一致的行为——这是一种很难处理的类型，因为它会因为误导信息而引起混乱。

1.1.2 拜占庭将军问题

在讨论分布式系统一致性问题之前，有必要回顾一下一致性机制的发展历程。

1962 年 9 月，Paul Baran 发表了一篇名为 *On distributed communications networks* 的论文，其中引入了加密签名。另外，在该论文中，还首次提出了去中心化网络这一概念。1982 年，Lamport 等人提出了一项思想实验，即一群领导拜占庭各部军队的陆军将军，计划从某一城市进攻或撤退。这里，将军们之间唯一的沟通方式是信使，他们需要就同时进攻达成一致意见，以赢得胜利。此处的问题是，一名或多名将军或许是叛徒，可以传达一个误导信息。因此，有必要找到一种可行的机制，让将军们之间达成某种协议，这样就可以同时进行攻击。相比于分布式系统，将军可视为节点，叛国者则视为拜占庭（恶意的）节点，而信使则是将军之间的沟通渠道。

该问题于 1999 年由卡斯特罗和利斯科夫提出，同时发布了可用的拜占庭容错(PBFT)算法。在 2009 年，首个实现方案随着比特币的出现而被制定。其中，Proof of Work (PoW) 算法形成了一种达成共识的机制。

1.1.3 一致性

在数据的最终状态上，一致性是指不信任节点之间的协议过程。为了达成共识，可以使用不同的算法。在两个节点之间（例如，在客户端-服务器系统中）达成协议相对简单；但是当多个节点参与到某一个分布式系统中，则需要在单值上达成一致，因而一致性机制难以实现。在多个节点之间达成一致性，这一概念称为分布式一致性问题。

1. 一致性机制

一致性机制定义为一组步骤，并被所有或大多数节点采用，以便在所提出的状态或值上达成一致。30多年来，这一概念一直被工业和学术界的计算机科学家所研究。最近，随着比特币和区块链的出现，一致性机制已经成为人们关注的焦点。

为了在一致性机制中提供所需的结果，必须满足各种需求，简述如下。

- ❑ 协议：全部可信节点指定同一值。
- ❑ 有效性：所有可信节点的商定值必须与至少一个可信节点所提议的初始值相同。
- ❑ 容错性：一致性算法应该能够在出现故障或恶意节点（拜占庭节点）的情况下运行。
- ❑ 完整性：作为一项要求，任何节点都不可做出多次决策。在单一的一致性循环中，节点仅可制定一次决策。

2. 一致性机制的类型

相应地，存在多种类型的一致性机制；一些常见的类型描述如下。

- ❑ 基于拜占庭式的容错：该机制不存在计算密集型操作（如部分哈希反转），这种方法依赖于一个简单的、发布签名消息的节点方案。最终，当收到一定数量的消息时，就会达成协议。
- ❑ 基于领导的一致性机制：需要节点实现领导选举机制，而获胜节点则提供结果值。

当前存在许多切实可行的方案，例如 Paxos——Leslie Lamport 于 1989 年提出的最为著名的协议。Paxos 节点分配了不同的角色，例如，申请人、接受者和学习者。同时，节点或进程被命名为副本，在存在故障节点的情况下，可在大多数节点之间达成一致。

RAFT 则是 Paxos 的替代方案，其工作方式可描述为：向节点赋予任意 3 种状态之一，即跟随者（Follower）、候选人（Candidate）或领导者（Leader）。领导者在候选者获取足够的选票后当选，并检查当前全部变化。如果完成了大多数跟随者节点的复制，那么领导者将提交所建议的全部更改。

从分布式系统的观点来看，更多关于一致性机制的细节内容已经超出了本章的范围，稍后将对此进行完整的介绍。具体的算法将在后续章节中讨论比特币和其他区块链时加以阐述。

1.2 区块链发展史

2008 年，随着比特币的出现产生了区块链这一概念，并在 2009 年予以实现。本章仅

对比特币加以简要介绍，后续章节还将对此进行深入讨论。缺少了比特币，区块链的历史发展进程将不再完整。

电子现金或数字货币的概念并不新鲜。自 20 世纪 80 年代以来，电子现金协议的存在基于 David Chaum 提出的模型。

1.2.1 电子现金

为了理解区块链技术，了解分布式系统的概念是必要的，电子现金这一概念对于理解区块链的成功应用来说也是至关重要的，如比特币，或广义的加密货币。分布式系统中的理论概念，如一致性算法，为比特币工作算法的具体实现提供了依据；此外，来自电子现金方案的各种理念也为加密货币的出现铺平了道路，特别是比特币。

本节将介绍电子现金这一概念，以及在加密货币之前即已存在的、与比特币发展紧密相关的各种概念。

1.2.2 电子现金的概念

电子现金系统需要解决的根本问题是责任性和匿名性问题。1984 年，David Chaum 通过引入两种加密操作，即盲签名和密钥共享，在其富有开创性的论文中解决了这两个问题。这些术语和相关概念将在第 3 章详细讨论。目前，有足够的理由说盲签名允许在没有亲身经历的情况下签署文档；而密钥共享则是一个概念，允许检测行为两次采用相同的电子现金令牌（重复付款）。

在此之后，还出现了诸如 Chaum、Fiat 和 Naor (CFN) 等其他协议，这些电子现金方案引入了匿名和重复付款检测。Brand 推出的电子现金则是在 CFN 上加以改进的另一个系统，且更富有效率；除此之外还引入了安全性规约（security reduction）这一概念，以证明电子现金方案的结算。安全性规约是一种加密中采用的技术，进而证明某种算法是安全的，并可通过另一个问题作为比较。换句话说，加密安全算法与其他难题一样难以破解；通过比较，可以推断出密码安全算法也是安全的。

作为一种控制电子垃圾邮件的 PoW 系统，Adam Back 于 1997 年引入了一种不同但相关的概念——哈希现金（hashcash）。这一想法很简单：如果合法的用户想要发送电子邮件，那么就需要计算一个 hash 作为证明——在发送电子邮件之前投入了合理的计算资源。生成 hashcash 是一个计算密集型过程，但并不妨碍合法用户发送电子邮件，因为合法用户需要发送的普通邮件数量可能较低。另一方面，如果一个垃圾邮件发送者想要发送电子邮件，对应数量通常是数千个，那么就不可能计算所有电子邮件的 hashcash，从

而使得垃圾邮件变得代价高昂；因此，当前机制可以用来阻止发送垃圾邮件。hashcash 需要消耗较多的计算资源，但是易于快速地进行验证，验证过程由接收电子邮件的用户执行。hashcash 是随着比特币挖掘过程中的应用而流行起来的。使用计算谜题或定价函数来防止电子垃圾邮件的想法最初是由 Cynthia Dwork 和 Moni Naor 在 1992 年提出的。定价函数是指在获得资源之前需要计算的硬函数名称。后来，Adam 在 1997 年提出了 hashcash，并引入了计算散列函数作为 PoW。

1998 年，Wei Dai 引入了 b-money，并提出了通过解决诸如 hashcash 这样的计算谜题来创造货币的想法。该理念基于对等网络，每个节点都维护自己的事务列表。

另一个类似的想法是由 Nick Szabo 提出的，他在 2005 年推出了 BitGold，并提议用数字货币来解决计算谜题。2005 年，Hal Finney 引入了加密货币的概念，将 b-money 和 hashcash 谜题结合在一起，但它仍然依赖于一个中心化的信任机构。

上述方案存在各种问题，某些问题尚不存在明确答案（节点间的不一致性），而另一些问题则依赖于中心受信的第三方和受信的时间戳。

2009 年名为比特币的加密货币首次实现，这是第一次在去信任的网络中解决了分布式协商一致的问题。它使用公钥加密和 hashcash 作为 PoW 来提供一种安全的、可控的、去中心化的数字货币。其中，较为重要的创新是有序列表，它由 PoW 机制的事务和密码保护组成。相关内容将在第 4 章中详细阐释。

通过考察上述相关技术及其发展史，即可看出电子现金方案和分布式系统之间是如何整合在一起来发明比特币的，现在则称作区块链。

相关内容如图 1.4 所示。

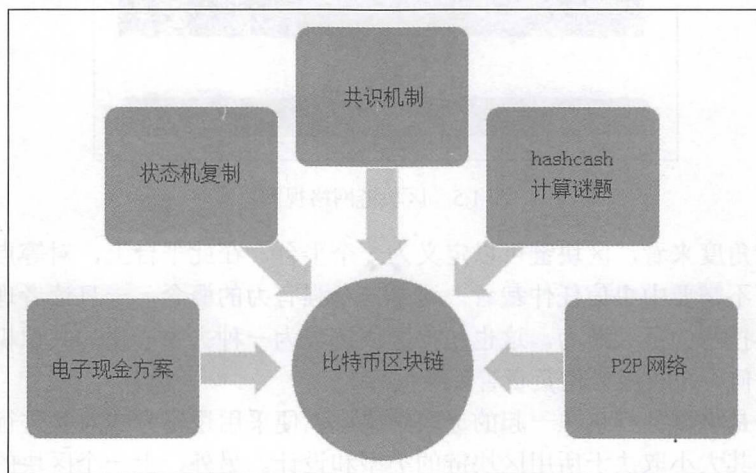


图 1.4 与比特币和区块链相关的各种概念

1.3 区块链简介

区块链存在多种定义方式，这取决于人们如何看待这一事物，其中包括商业角度和技术角度。

区块链的核心是一个点对点的分布式账本，此类账本具有加密安全性，且仅可追加内容，同时不可更改（很难更改），只有在对等身份成员之间达成共识或协议时才能更新。

区块链可视作在互联网上运行的、分布式 P2P 网络中的一层，类似于在 TCP/IP 上运行的 SMTP、HTTP 或 FTP，如图 1.5 所示。

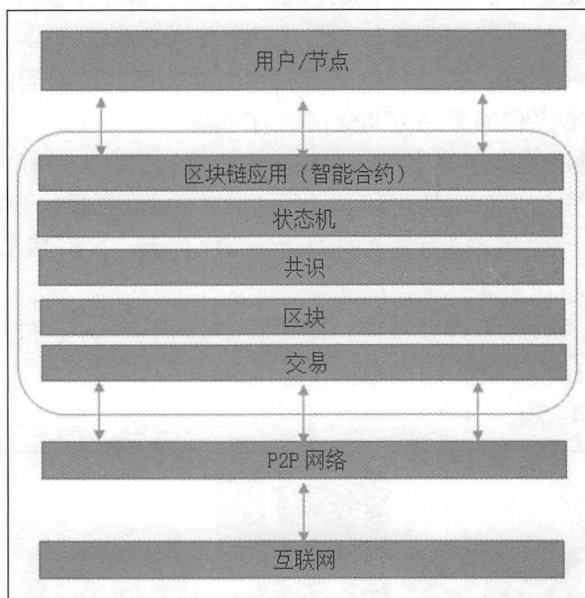


图 1.5 区块链网络视图

从业务的角度来看，区块链可以定义为一个平台，在此平台上，对等点可以通过事务交换值，而不需要中央信任仲裁者。这是一个强有力的概念，一旦读者理解它，就会意识到区块链技术的巨大潜力。这也使得区块链成为一种去中心化的共识机制，在该机制中，没有任何一个权威机构负责管理数据库。

区块仅仅是将事务打包在一起的选取结果，以便采用逻辑方式对其进行组织。区块由事务组成，其大小取决于所用区块链的类型和设计。另外，上一个区块的引用包含当前区块中，除非该区块是一个创始区块。创始区块是区块链中的第一个区块，在区块链

开始时即被硬编码。同时，区块的结构也取决于区块链的类型和设计，但通常有一些属性是区块功能不可或缺的，如区块头（指向前一区块）、时间戳、nonce、柜台交易、交易和其他属性。

图 1.6 显示了简单区块的示意图，图中展示了某一区块的一般性描述，与区块链技术对应的特定区块结构将在后续章节中加以深入讨论。

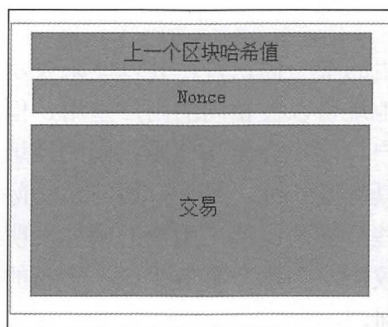


图 1.6 区块结构

1.3.1 区块链技术的各种定义

区块链是一种去中心化的协商一致机制。在区块链中，所有的对等点最终会达成关于交易状态的协议。

区块链是一个分布式共享账本，可以看作是一个共享的交易总账。事务经排序后被分组为区块。目前，现实世界的模型基于每个组织维护的私有数据库。对于使用区块链的全体成员组织，分布式账本可视为一个单一的来源。

作为一种数据结构，区块链基本上是一个使用散列指针而非普通指针的链表。其中，哈希指针被用来指向前一个区块。

图 1.7 显示了一般的区块链结构。

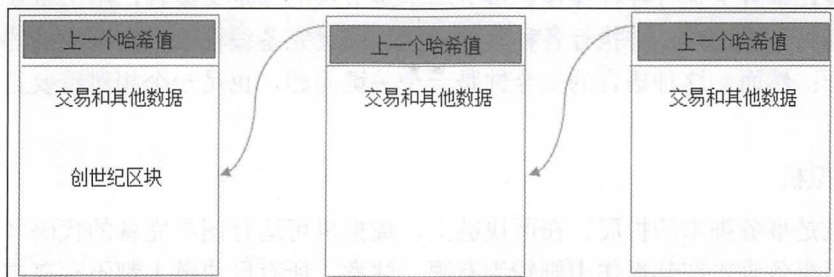


图 1.7 常规区块链结构

1.3.2 区块链中的一般元素

本节仅定义了区块链中的一般元素，更精确的元素将在后续章节中讨论各自的区块链时予以解释，如以太坊区块链。

1. 地址

地址是区块链事务中使用的唯一标识符，并以此来表示发送者和收件人，通常表示为公钥或派生自公钥。虽然地址可以被相同的用户重用，但地址本身是唯一的。然而，在实际操作过程中，单个用户可能不会再次使用相同的地址，并为每个事务生成一个新的地址，这个新生成的地址具有唯一性。另外，比特币实际上是一个匿名系统。终端用户通常无法直接识别，但一些反匿名比特币用户的研究表明，用户仍可被成功识别。作为一种良好的实践方式，建议用户为每个事务生成一个新的地址，防止将事务链接到公共所有者，从而避免身份识别。

2. 事务

事务是区块链的基本单元，表示地址间的值传输。

3. 区块

区块由多个事务和一些其他元素组成，例如，上一个区块哈希值（哈希指针）、时间戳和 nonce。

4. P2P 网络

顾名思义，P2P 网络是一种网络拓扑结构，所有的对等点之间可以彼此通信、发送和接收消息。

5. 脚本或程序语言

该元素在事务上执行各种操作。事务脚本是节点的预定义集合，用于将令牌从一个地址转移到另一个地址，并执行各种其他功能。图灵完备编程语言是区块链的一个较为理想的特征；然而，这种语言的安全性是一个关键问题，也是一个相对重要且正在进行研究的领域。

6. 虚拟机

虚拟机是事务脚本的扩展。在区块链上，虚拟机可运行图灵完备的代码（作为智能合约），而事务脚本在其操作中则较为有限。注意，所有区块链上都不存在虚拟机。然而，各种区块链使用虚拟机运行程序，如以太坊虚拟机（EVM）和链虚拟机（CVM）。

7. 状态机

一个区块链可以被看作是一个状态转换机制，在该机制中，一个状态从初始状态被调整为下一个状态，并以节点的事务执行和验证过程的结果作为最终形式。

8. 节点

区块链网络中的一个节点根据其所扮演的角色执行各种功能。一个节点可以生成并验证事务，同时进行挖掘以促进一致性关系并确保区块链的安全性。这是通过遵循协商一致的协议来完成的（最常见的是 PoW）。除此之外，节点还可以执行其他功能，如简单的支付验证（轻量级节点）、验证器以及许多其他功能，这取决于所使用的区块链的类型以及分配给该节点的角色。

9. 智能合约

智能合约运行在区块链之上，并在满足某些条件时封装执行的业务逻辑。区块链并不支持智能合约特性，但考虑到智能合约提供给区块链应用程序的灵活性和各项功能，现在其已经成为一个非常被期望的特性。

1.3.3 区块链特性

区块链提供了多种特性，如下所示。

1. 分布式一致性

分布式协商一致是区块链的主要基础，这使区块链能够呈现唯一版本的事实结果，在缺少中心信任机制授权的情况下，各方都能达成一致。

2. 事务验证

从区块链节点上发布的任何事务都基于预先确定的规则集进行验证，并且只选择有效的事务包含在一个区块中。

3. 智能合约平台

区块链定义为一个平台，程序可以运行于其上，并代表用户执行业务逻辑。正如前面所解释的，并不是所有的区块链都具备执行智能合约的机制；然而，这是一个令人期待的特性。

4. 在对等点间传输值

区块链允许通过令牌在用户之间传递值。相应地，令牌可视作数值的载体。

5. 生成加密货币

取决于所使用的区块链的类型，该内容是一个可选的特性。区块链可以生成加密货币，以激励那些验证交易并投入资源以确保区块链安全的矿工。

6. 智能属性

首先，可以采用一种不可撤销的方式将数字或实物资产与区块链联系在一起，这样任何人都无权占有，资产完全由个人控制，且无法被重复使用或双重拥有。例如，数字音乐文件可以多次复制而不受任何控制；相比较而言，在区块链上，没有人可独自占有该文件，除非决定将其转让给某人。这一特点对数字版权管理（DRM）和电子现金系统有着深远的影响，在这些系统中，双支出检测是一个关键的要求。双重消费问题最初是用比特币解决的。

7. 安全提供商

区块链基于已证实的加密技术，以确保数据的完整性和可用性。一般来说，保密性之所以受到限制是源自透明度这一要求。对于金融机构和其他需要隐私和交易机密的行业而言，这已经成为适应性的主要障碍。正因为如此，这一领域的研究也处于较为活跃的状态，而且已经取得了一些良好的进展。可以认为，在许多情况下，保密性并不是真正需要的，而透明度则是首选，例如比特币。然而，在某些情况下保密性依然是可取的。这一领域的研究已经非常成熟，在区块链提供保密和隐私方面已经取得了重大进展。最近的一个例子是 Zcash，后续章节将对此进行详细讨论。除此之外，区块链还提供了其他安全服务，如认可和身份验证功能，因为所有的操作都是通过私钥和数字签名来保护的。

8. 不变性

不变性是区块链的另一个关键特征，也就是说，在区块链中添加的记录是不可变的。回滚变化内容基本无法实现，该行为需要一种难以负担的计算资源。例如，如果一个恶意的用户想要改变之前的区块，就需要对已经添加到区块链的所有区块重新计算 PoW，其难度使得区块链上的记录几乎是不可变的。

9. 唯一性

区块链的这一特性确保了每个事务都是唯一的，且尚未被消费。这一点在加密货币中尤为重要，因为在这些货币中，要想发现并避免双重支出，唯一性将是一个关键因素。

10. 智能合约

区块链提供了一个平台并可履行智能合同，并可视作位于区块链上的自主程序，可封装业务逻辑和代码，以便在满足某些条件时执行所需的功能。这确实是区块链的一个

革命性的特征，同时实现了灵活性、可编程性，以及对于特定业务需求执行的操作，区块链用户可对此加以控制。

1.3.4 区块链技术应用

区块链技术在各个领域都有大量的应用，包括但不限于金融、政府、媒体、法律和艺术等。第 9 章将讨论各个行业中的实际用例。现在可以说，区块链的潜力及实施承诺体现在几乎所有的行业中，并已经开始或即将踏上从区块链技术中获益的旅程。

本节讨论创建区块的一般方案，并概述区块的生成方式，以及事务与区块之间的关系。

(1) 节点通过私钥签名来启动事务。

(2) 针对对等点，使用 Gossip 协议来传播事务，该协议根据预置标准对事务进行验证。通常，需要不止一个节点来验证事务。

(3) 一旦事务被验证，即被包含在一个区块中，随后将传播至网络。在这一点上，事务将被确认。

(4) 新创建的区块现在变成了账本中的一部分内容，而下一个区块将以加密方式将其链接回该区块。其中，对应的链接是一个散列指针。在这个阶段，事务得到第二次确认，而当前区块则涵盖了首次确认内容。

(5) 接下来，在每次创建新区块时重新确认事务。通常，需要在比特币网络中进行 6 次确认才能考虑交易的最终结果。

步骤 (4) 和步骤 (5) 可以认为是非强制性操作，因为事务本身在步骤 (3) 中最后确定；但是，如果需要区块确认，以及进一步的事务重新确认将在步骤 (4) 和步骤 (5) 中执行。

1.3.5 区块链发展层次

本节将讨论不同层次的区块链技术。可以想象，由于区块链技术的快速发展和进步，许多应用程序将随着时间的推移而发生变化。其中，一些应用可能已付诸实现，而一些应用可以根据区块链技术的当前发展速度来设想其未来趋势。

首先，Melanie Swan 在其撰写的 *Blockchain, Blueprint for a New Economy* 一书中描述了 3 种级别，并根据每个类别的应用程序划分为区块链。除此之外，稍后还将讨论 Tier X 或 Generation X。当区块链技术足够先进时，Melanie Swan 认为这种层级结构终将成为现实。

1. 区块链 1.0

区块链 1.0 是随着比特币的发明而引入的，基本用于加密货币。此外，由于比特币是第一个加密货币的具体实现，因此区块链技术的第 1 代分类仅包括加密货币是有意义的。所有可替换硬币和比特币都属于这一类，其中包括核心应用程序，如支付和应用。

2. 区块链 2.0

区块链 2.0 主要用于金融服务，在这一阶段中引入了合同。这包括各种金融资产，例如衍生品、期权、互换和债券。超出货币、金融和市场之外的应用程序也将纳入这一层。

3. 区块链 3.0

区块链 3.0 用于金融服务行业之外，同时还包括政府、卫生、媒体、艺术和司法等更综合的行业。

4. Generation X（区块链 X）

区块链 X 可视为区块链发展过程中的一个“奇点”，有一天我们将会拥有一个公共区块链服务，任何人都可以使用它，就像谷歌搜索引擎一样，进而在社会的所有领域中提供服务。这将是一个公开的开放式账本，具备运行于区块链上的、一般用途的“理性认知主体”（引自 *Machina Economicus*），制定决策并与其他智能认知主体（代表人类）进行交互，通过代码而不是法律或纸质契约加以管理。这将在第 13 章详细阐述。

1.4 区块链类型

基于区块链在过去几年的发展方式，可以将其划分为不同的类型，但在某些场合下也存在部分重叠的属性。

1.4.1 公有区块链

顾名思义，这一类型的区块链对公众开放，任何人都可以作为决策过程中的节点参与进来。用户的参与不一定会得到奖励。另外，任何人均不会持有该账本，而且向任何参与者开放。所有未受许可的账本用户都在本地节点上维护账本的副本，并使用分布式协商一致机制来决定最终的账本状态。这一类区块链也被称为“无许可”账本。

1.4.2 私有区块链

这一类型的区块链体现了一定程度上的私有性，仅向财团、团体或组织开放，进而

确定账本的共享过程。

1.4.3 半私有区块链

半私有区块链兼具私有性和公有性。其中，私有部分由某一团体控制；而公有部分则向参与者开放。

1.4.4 侧链技术

更准确地讲，这一概念应称作楔入式侧链，货币将据此在区块链之间移动。常见的应用包括山寨币（altcoin，可替换的加密货币），并作为某项权益证明记录货币。侧链存在两种类型，相关示例包括单路楔入式侧链以及双路楔入式侧链。针对后者，货币可在主链和侧链间移动，并在必要时返回至主链。

1.4.5 许可账本

许可账本可视为一个区块链，该网络的参与者是已知且受信的。许可账本不需要使用分布式的协商一致机制；相反，关于区块链记录状态，可以使用一致性协议来维护共享版本。这里，经许可后的区块链无须制定为私有链——可表示为公有区块链，但会涉及一定的访问控制权限。

1.4.6 分布式账本

顾名思义，分布式账本分布在参与者之间，并扩散于多个站点或组织中。这一类型的账本可以是私有链或公有链。关键思想是，与许多其他类型的区块链不同，对应记录是连续存储的，而不是按区块排序。这一概念被用在 Ripple 上。

1.4.7 共享账本

共享账本是一个相对通用的术语，用于描述公众或财团共享的应用程序或数据库。

1.4.8 全私有和专有区块链

这一类区块链可能缺少主流应用，皆因偏离了区块链技术去中心化这一核心思想。尽管如此，在一个组织内部的特定私有环境中，仍然可能需要共享数据，并提供一定程

度的数据真实性保证。在这种情况下，这一类区块链依然可用。例如，在多个政府部门之间进行协作和共享数据。

1.4.9 标记化区块链

这一类型的区块链可视为标准的区块链，通过挖掘或初始分布来生成加密货币，并作为一致性处理结果。

1.4.10 无代币区块链

无代币区块链并非是真的区块链，其原因在于缺少价值转移的基本单位；但在不需要于节点间传递值的情况下仍然很有价值，而且只需要在受信各方之间共享某些数据。

在 1.4.11 节中，将从区块链角度讨论共识这一概念。共识机制是区块链的主要内容，通过一个可选的处理过程（称为挖掘）来实现对控制的去中心化操作。同时，共识算法的选择也由所用的区块链类型管控。需要注意的是，并不是所有的共识机制均适用于全部区块链类型。例如，在公共许可程度较低的区块链中，使用 PoW 则更具意义，而不是基于权威证明的基本协议机制。因此，需要针对某个区块链项目选择适宜的共识算法。

1.4.11 区块链中的共识

共识（一致性）基本上是一个分布式的计算概念并用于区块链中，进而提供一种方法，以使区块链网络上的所有成员提供单一的承兑方式。

大致来讲，存在下列两种共识机制分类：

❑ 领导和最终价值的产生方式基于证据或中本聪一致性原则。

❑ 基于拜占庭容错机制，这一种传统的多轮投票方案。

稍后将给出现有的一些算法，以及某些仍处于研究中的算法。相关内容并未面面俱到，只是试图解释其中的某些重要算法。

1. 工作量证明机制

这种类型的共识机制依赖于如下证明：在提出一个网络接受价值之前已经花费了足够的计算资源。该机制用于比特币和其他加密货币。目前，这是唯一一种对 Sybil 攻击取得巨大成功的算法。

2. 权益证明

该算法基于如下理念：节点或用户在系统中具有足够的权益。例如，用户已在系统

中投入了足够的资金，因此，恶意行为获得的收益将超出执行系统攻击。这一概念首先由 Peercoin 提出，并应用于以太坊区块链中。权益证明（PoS）中另一个较为重要的概念是币龄，且源自尚未消费的时间量和货币量。在该模型中，生成并签订下一个区块的机会随着币龄而增加。

3. 委托权益证明

委托权益证明（DPOS）是对标准 PoS 的一种创新，在此基础上，系统中的每个节点都可以通过投票将事务的验证委托给其他节点。该机制用于比特币中。

4. 流逝时间证明

该算法由 Intel 提出，并使用可信执行环境（TEE）通过可靠的等待时间在领导选举过程中提供随机性和安全性。另外，算法需要使用到 Intel SGX（软件保护扩展）处理器，以提供安全保证。这一概念将在第 9 章的 Intel Sawtooth Lake 区块链项目中加以讨论。

5. 基于保证金的共识

希望加入网络中的节点在生成区块链之前需要存入安全保证金。

6. 重要性证明

该算法十分重要且不同于权益证明。重要性证明不仅依赖于用户在系统中所拥有的权益份额，而且还将监视用户对令牌的使用和移动，以建立信任和重要性级别。该算法主要用于 Nemcoin 中。

7. 联邦共识和联邦拜占庭共识

该协议中的节点使用在恒星共识协议中，保留一组公开信任的对等点，并只传播由大多数受信节点验证的事务。

8. 信誉机制

顾名思义，领导者是基于在网络上建立的声誉而当选的，该过程可以通过其他成员的投票而实现。

9. 实用的拜占庭容错机制

实用的拜占庭容错（PBFT）实现了状态机的复制，并对拜占庭节点提供了容错机制。其他各种协议，包括但不限于 PBFT、PAXOS、RAFT 和联邦拜占庭协议（FBA），也被用于或被提议用于各种分布式系统和区块链的实现中。

1.5 CAP 定理和区块链

令人感到奇怪的是，区块链似乎违背了 CAP 定理，尤其是在最成功的比特币实现中，但事实并非如此。在区块链中，一致性将被削弱，进而支持可用性和分区容错性机制。在这种情况下，区块链上的一致性（C）并不是与分区容错性（P）和可用性（A）同时实现的，而是随着时间的推移而实现的。该过程被称作最终一致性，这种一致性是由多个节点通过一段时间内的验证而实现的。为此，在比特币中引入了“采矿”这一概念，该处理过程使用了称为 PoW 的共识算法以实现协商一致性过程。在更高的级别中，可以将挖掘定义为某个处理操作，并用于向区块链中添加更多的区块。

1.6 区块链的优点和局限性

区块链的优点在业界被广泛地提及，相关人士也对此进行了总结，大致包含了以下内容。

1. 去中心化

去中心化是区块链的核心概念以及显著优点，也就是说，不需要守信的第三方或中介来验证交易；相反，一种共识机制被用来就交易的有效性达成一致协议。

2. 透明度和信用机制

由于区块链具有共享特征，因而每个人都可以看到区块链中的内容，这也使得系统处于透明状态，从而建立了信任机制。这在某些情况下更有实际意义，例如，支付资金或福利机制。其中，个人自由裁量权应受到一定的限制。

3. 不变性

一旦将数据写入区块链，就很难将其更改回来。实际上，数据并不是真正意义上的“不可变”，但更改数据非常困难，几乎不可能实现。因此，在维护一个不可变的事务账本时，不可变性则是一个显著的优点。

4. 高可用性

由于当前系统基于对等网络中的数千个节点，并且在每个节点上复制和更新数据，因而系统具有可用性。即使节点脱离网络或无法访问，但网络作为一个整体仍可继续工

作，从而具备了一定的高可用性特征。

5. 高安全性

在区块链上的所有事务均经过加密处理，并具有一定的完整性。

6. 简化范式

目前，金融或健康等许多行业中的模型是相当混乱的，由于系统性质间的差异性，多个实体维护自己的数据库和数据共享变得非常困难。但是，由于一个区块链可以作为团体间的一个独立的共享账本，因而模型可得到适当的简化，即降低每个实体所维护的、独立系统的复杂度。

7. 快速的交易过程

在金融行业中，特别是在交易后的结算功能中，区块链扮演着至关重要的角色——可提升交易后的结算速度，期间不需要一个漫长的等待过程，其中包括验证、对账、清算等，因为数据协议的单个版本已经存在于金融机构之间的共享账本中。

8. 节约成本

在区块链模型中，不需要第三方或结算公司，这可以极大地削减支付给计算公司或受信任第三方的费用。

1.7 区块链技术的限制和挑战

与任何技术一样，为了确保系统的健壮性、可用性和易于访问性，人们仍面临着一些挑战性任务，区块链技术也不例外。事实上，学术界和业界正在努力克服区块链技术所面临的种种困难，其中包括以下方面：

- ☐ 可扩展性。
- ☐ 适应性。
- ☐ 相关规则的制定。
- ☐ 某些相对不成熟的技术。
- ☐ 隐私性。

相关内容将在第13章加以讨论。

本章内容具有概述性质，且较少涉及具体技术。在第3章介绍了密码学之后，将在适当的技术深度和细节上讨论具体的区块链解决方案。

1.8 本章小结

本章向读者介绍了区块链技术。首先讨论了分布式系统的一些基本思想，以及区块链的历史，并解释了电子货币和 `hashcash` 等概念。此外，还提出了不同区块链的各种定义，简要介绍了区块链技术的一些应用，同时还解释了不同类型的区块链。最后，本章阐述了区块链这一新技术的优缺点。其中，某些主题只是予以简单介绍，后续章节中还将对此加以深入讨论，例如，区块链技术的挑战和限制，但并未提供相关的细节内容。第 2 章将会探讨去中心化概念，这也是区块链及其大量应用的核心内容。

第2章 去中心化

去中心化并不是一个全新的概念。长期以来，这一理念一直被用于战略、管理和治理方面。去中心化的基本思想是将控制权和职权下放至外围，而不再是由一个中央集权控制整个组织。这种机制也为旗下组织带来了一些益处，例如效率的提升，快速的决策过程，以及更好的激励制度，同时也减少了高层管理的负担。

在本章中，去中心化概念将在区块链的背景下讨论——两者的目标是相似的，即不存在可参与控制的独立中央权威机构。本章还将通过相关示例介绍去中心化的具体方法，以及操作流程。此外，还将详细讨论区块链生态系统的去中心化、去中心化应用程序以及去中心化平台。许多令人兴奋的应用和理念均已出现在分散的区块链技术中，本章将对此予以介绍。

2.1 基于区块链的去中心化

去中心化是区块链技术提供的核心利益和服务。区块链的设计是一个完美的载体，同时提供了一个不需要任何中介的平台，并且可以通过共识机制选择许多不同的领导者。这种模式允许任何人竞争成为决策权威。这一竞争行为通过一个共识机制加以管理，最常用的方法就是工作量证明（PoW）。

去中心化应用包括半去中心化以及完全去中心化，且程度不一，这取决于具体的需求和环境。我们可以从区块链的角度来看待去中心化。作为一种机制，它提供了一种方法来重新建模现有的应用程序和范例，或者构建新的应用程序，以便完全控制用户。

信息和通信技术（ICT）传统上基于一种集中式的模式，即数据库或应用程序服务器处于中央权威的控制之下，例如系统管理员。随着比特币和区块链技术的出现，这种模式已经发生了变化。当今技术可以让任何人启动一个去中心化系统（并在没有单点故障或单一信任权限的情况下对其进行操作）。该系统可自动运行，也可以根据在区块链上运行的、去中心化应用程序中使用的治理类型和模型来要求人工干预。

图 2.1 显示了现有的不同类型的系统，即中央、分布式和去中心化系统。这一概念最早是在 1964 年由 Paul Baran 在通信网络中的分布式通信网络上提出的。

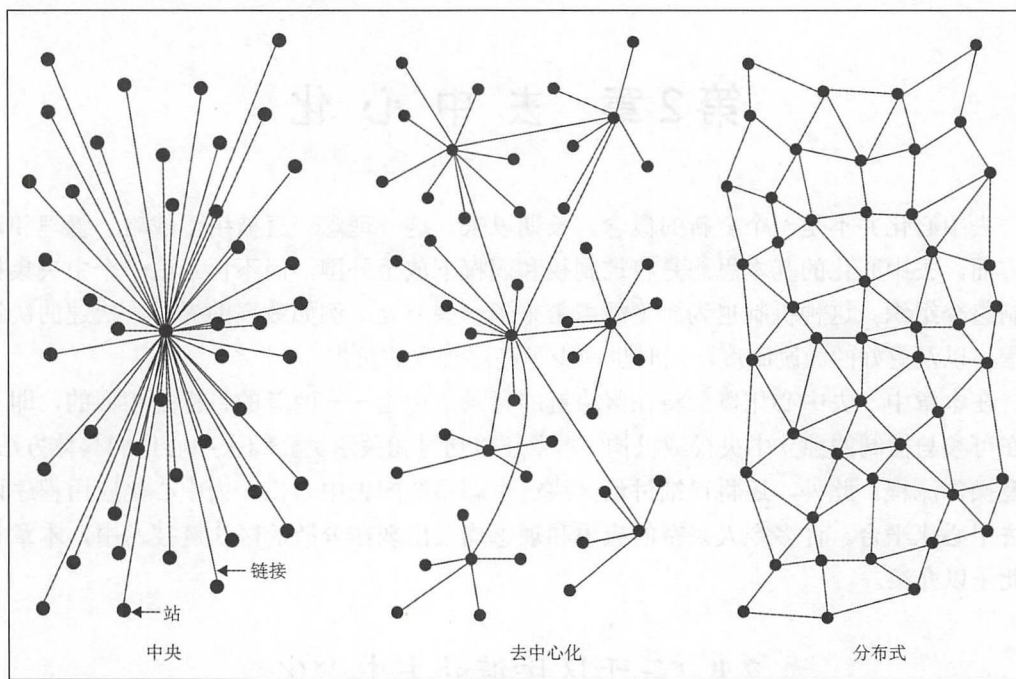


图 2.1 不同的网络/系统类型

集中式系统是传统的（客户机-服务器）IT 系统，其中包含了控制系统中的单一权威，完全负责系统上的所有操作。中央系统的所有用户都依赖于单一的服务源。在线服务提供商如 eBay、谷歌、亚马逊、苹果应用商店以及大多数其他提供商都使用这种公共服务模式。另一方面，在分布式系统中，数据和计算分布在网络中的多个节点上。有些时候，这一术语常与并行计算混淆。虽然其间存在一个重叠的定义，但两个系统之间仍存在一些差别：在并行系统中，计算是由所有节点同步执行；而在分布式系统中，计算不可能以并行方式执行，数据仅可在用户视为单一聚合系统的多个节点上进行复制。这两种模型都使用了不同的方法来实现容错和速度调整。在这一类模型中，中央权威机构控制所有节点并管理处理过程。这意味着系统在本质上仍然是中心化的。

去中心化系统和分布式系统之间的关键区别在于，在分布式系统中，仍然存在管理整个系统的中央权威机构；而在去中心化系统中，该机构将不复存在。去中心化系统是节点不依赖于单个主节点的网络类型；相反，控制权分布在许多节点中。例如，这与企业中的组织结构类似，每个部门都有自己负责的数据库服务器模型，进而从中央服务器中移除权利，并将其分发给管理自己数据库的子部门。

在去中心化范例中，应用创新行为主要体现在去中心化的共识机制，并随着比特币

的出现而兴起。这使得用户可以通过共识算法，在不需要中央可信的第三方、中介或服务提供者的情况下达成一致。

2.2 去中心化方法

相应地，存在两种方法可实现去中心化操作，下面几个小节将对此加以讨论。

2.2.1 非中介化

这一概念可以通过一个例子来解释。假设用户想把钱寄给另一个国家的朋友，一般需要到银行办理转账业务。在这种情况下，银行维护一个更新后的中央数据库，并确认转账成功。当采用区块链技术时，该过程将大大简化。用户只需知晓朋友在区块链上的地址即可。这样，就不再需要中介了，而去中心化则是通过非中介化实现的。然而，鉴于严格的监管和规章制度，金融部门去中心化实现的方式依然存在争议。无论如何，这种模式不仅可以用于金融行业，也可以用于许多其他行业。

2.2.2 竞争

在这种方法中，一组服务提供者相互竞争，以便被系统选择进而提供有效的服务。这种模式并没有实现完全的去中心化，但在一定程度上可保证中介或服务提供者不再垄断服务。在区块链技术的背景下，可假定存在一个系统，在这个系统中，智能合约可以根据声誉、评分、评论和服务质量，从大量的提供者中选择一个外部的数据提供者。当然，这不会生成完全的去中心化结果，但却允许智能合约根据前面提到的标准做出自由选择。这样一来，服务提供者之间就形成了竞争环境，进而在彼此竞争中被选择为数据提供者。

图 2.2 显示了不同级别的去中心化结果。其中，左图表示为一种传统方法，即中央控制系统；右图实现了完全非中介化过程；中图展示了相互竞争的中介或服务提供者。相应地，中介或服务提供者根据声誉或投票加以选择，从而实现部分去中心化过程。

虽然去中心化过程包含许多优点，例如，透明度、效率、节约成本、受信生态系统的开发，以及在某些情况下的隐私和匿名性，但往往也会面临某些挑战，如安全需求、软件缺陷和人为错误。例如，在一个去中心化系统中（如比特币或以太坊），安全问题通常是由私钥予以保障的。但如何确保与私钥关联的智能财产之间的有效性？例如，如果由于人为错误丢失私钥，或者由于智能合约代码中的缺陷，去中心化应用程序易受到

外来攻击。在开始使用区块链和去中心化应用程序之前，读者需要理解以下观点：不是所有的事务都是需要（或者可以被）去中心化的。

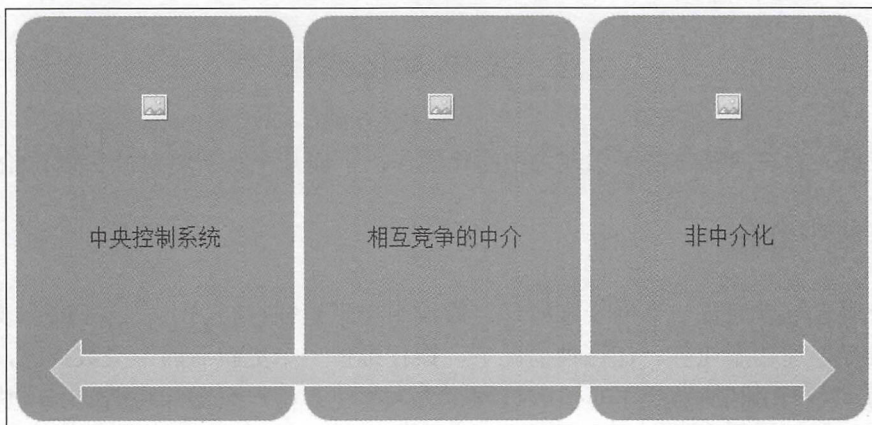


图 2.2 去中心化的规模

2.3 去中心化流程

尽管在比特币或区块链之前也存在着一些系统，并在某种程度上可归类去中心化系统，如 BitTorrent 或 Gnutella 文件共享，但随着区块链技术的出现，许多新方案正在考察中，以利用这种新技术实现去中心化操作。通常，比特币区块链是人们的首选方案，因为它已经被证明是最富有活力和安全性的区块链，市值接近 120 亿美元。另一种方案是使用其他区块链，如以太坊，目前也是许多去中心化应用程序开发人员的选择。

1. 去中心化的实现方式

Arvind Narayanan 等人提出了一个框架，可以用来评估在区块链技术环境下各种事物的去中心化需求。该框架基本上提出了 4 个问题，进而可对去中心化需求条件进行评估。相关问题如下所示：

- ☐ 去中心化的含义。
- ☐ 去中心化所需要的级别。
- ☐ 采用哪一种区块链。
- ☐ 采用哪一种安全性机制。

第一个问题简单地询问什么是去中心化系统。去中心化系统可以是任何系统，例如一个认证系统或交易系统。第二个问题可以通过之前讨论的去中心化规模来指定相应的

级别——可以是完全非中介化或部分非中介化。第三个问题非常简单，开发人员可以选择哪个区块链适合特定的应用程序，例如，比特币区块链、以太坊区块链，或其他任何适合特定应用程序的区块链。最后，对于去中心化系统的安全性问题，需要对安全机制提出一个关键问题。例如，对应机制可以是原子性的，即事务全部执行或不执行，这可确保系统具有一定的完整性。其他机制还包括声誉机制，以支持系统中不同的受信度。

2. 示例

本小节将考察上述框架的应用示例。

第一个例子将讨论一个资金转移系统，并需要实现去中心化过程。在这种情况下，可以回答前面提到的以下 4 个问题，并对去中心化需求条件进行评估。

- ☐ 资金转移系统。
- ☐ 非中介化（去中介化）。
- ☐ 比特币。
- ☐ 原子机制。

通过回答上述 4 个问题，可以看出支付系统是如何实现去中心化的。在此基础上，通过去中介化，即可实现去中心化的资金转移系统，并在比特币区块链上实现基于原子性的安全保证。

同样，该框架也可以用于其他系统中，且需要对去中心化进行评估。通过回答这 4 个简单的问题，就可以很清楚地知道如何采取措施实现去中心化系统。

2.4 区块链和完整的生态圈去中心化操作

为了实现完整的去中心化结果，还需针对区块链周围环境执行去中心化操作。区块链本身就是一个在常规系统上运行的分布式账本，相关元素包括存储、通信和计算以及其他各项因素，如身份和财富，传统上也是基于集中范例。为了实现一个完全去中心化的生态系统，也需要针对这一类因素执行去中心化操作。

2.4.1 存储

数据可以直接存储在区块链中，这样即实现了去中心化操作，但这种方法也存在一个缺点：区块链不适合存储大量的数据。也就是说，区块链可以存储简单的事务和某些数据，但肯定不适合存储图像或大型数据对象，这一点与传统的数据库系统十分类似。对此，一种更好的选择方案是使用分布式哈希表（DHT）。DHT 最初用于 P2P 文件共享

软件，如 BitTorrent、Napster、Kazaa 和 Gnutella。DHT 研究领域随着 CAN、Chord、Pastry 和 Tapestry 项目的出现变得十分流行。BitTorrent 是一类颇具规模的高速网络，但问题是用户并不会无限期地保存相关文件。如果节点脱离了数据网络，则该节点将无法被检索到，除非该节点再次加入网络中，以使文件再次可用。这里的两个重要因素是指高可用性和链接稳定性，这意味着在需要的时候数据应该可用，而且网络链接也应该总是可以访问的。Juan Benet 的 Inter Planetary File System (IPFS) 即同时拥有这两种属性，旨在通过替换 HTTP 协议提供一个去中心化的万维网。IPFS 使用 Kademlia DHT 和 merkle DAG（有向无环图）分别提供存储和搜索功能。

激励机制基于一种名为 Filecoin 的协议，该协议向使用 BitSwap 机制存储数据的节点提供激励。BitSwap 机制允许节点保留一个简单的字节账本（在一对一关系下发送字节或接收字节）。另外，在 IPFS 中使用基于 Git 的版本控制机制，可对数据版本提供相应的结构和控制。

除此之外，还存在诸如 Ethereum 蜂群，storj 和 maidsafe 等其他选择方案。其中，Ethereum 包含自身的去中心化和分布式的生态系统，并使用集群存储和 whisper 协议进行通信；而 maidsafe 则致力于提供一个去中心化的万维网。所有这些项目将在后续章节中详细地讨论。

BigChainDB 是另一个存储层去中心化项目，目的是提供一个可伸缩、快速和线性可伸缩的去中心化数据库，而不是传统的文件系统。BigChainDB 对去中心化的处理平台和文件系统进行了有益的补充，如 Ethereum 和 IPFS。

2.4.2 通信

一般认为，Internet（区块链中的通信层）是去中心化的，这在某种程度上是正确的，因为互联网最初旨在建立一个去中心化的系统。电子邮件和在线存储等服务当前都基于某种模式，即服务提供者处于控制之中并得到了用户的信任，以在必要时向其提供服务功能。该模型基于中央权威（服务提供者）的信任，用户并不负责管理其数据；甚至密码也可存储在受信的第三方系统上。另外，有必要以某种方式向个别用户提供管理权限，以确保用户可以访问其数据，并且不依赖于第三方。互联网接入（通信层）一般来自互联网服务提供商（ISP），它也是互联网用户的中心枢纽。如果 ISP 因为政治或其他原因被关闭，那么该模型中的通信也随之中断。另一种选择是使用网状网络。与 Internet 相比，尽管网状网络的功能有限，但仍然提供了一种去中心化的替代方法。其中，节点间可以直接对话，而不需要像 ISP 那样采用中心枢纽。



注意:

网格网络的一个例子是 Firechat (<http://www.opengarden.com/firechat.html>), 并允许 iPhone 用户在没有互联网的情况下以点对点的方式直接进行通信。

假设存在一种网络, 用户可借助该网络控制他们的通信, 没有人能因为政治或审查原因关闭该网络。这可能是在区块链生态系统中去中心化通信网络的下一个发展阶段。必须指出的是, 这种模式可能只会存在于司法领域内, 其间, 互联网受到政府的审查和控制。

如前所述, 互联网最初旨在建立一个去中心化的网络。然而, 多年以来, 随着谷歌、亚马逊和 eBay 等大型服务提供商的出现, 控制权逐渐转向了这一类商业巨头的手中。例如, 电子邮件的核心即是一个去中心化的系统, 任何人都可以方便地启用这项服务, 并接收、发送电子邮件。即使如此, 仍然存在一类更好的选择方案, 并已开始向终端用户提供管理服务。自然地, 考虑到便捷性和免费等因素, 人们也倾向于选择这一类中心化服务。然而, 免费服务却是以昂贵的个人数据为代价的, 许多用户并没有意识到这一点。同时, 这也是一个展示互联网如何走向集权的例子。区块链又一次把这种权力下放到人们手中, 这项技术现已得到大力推广, 人们也期待着从中获取更大的收益。

2.4.3 计算

计算或处理过程的去中心化行为也是通过区块链技术实现的, 例如以太坊。其中, 包含业务逻辑的智能合约可以运行于该网络上。其他区块链技术也提供了类似的处理层平台, 业务逻辑可以以去中心化的方式在网络上运行。

图 2.3 显示了去中心化的生态系统概览。在底层, Internet 或网格网络提供了一个去中心化的通信层; 随后, 存储层使用诸如 IPFS 和 BigChainDB 这一类技术以对去中心化加以支持; 最后, 区块链表示为一个去中心化的处理层。另外, 区块链可以以有限的方式提供一个存储层, 但这严重阻碍了系统的速度和容量。因此, 诸如 IPFS 和 BigChainDB 之类的其他解决方案更适合以一种去中心化的方式存储大量数据。图 2.3 中顶部显示了 Identity 和 Wealth 层。互联网上的身份是一个非常大的话题, 像 bitAuth 和 OpenID 这一类系统均提供了身份验证和身份识别服务, 且具有不同程度的去中心化和安全性假设条件。

区块链能够为各种问题提供解决方案。Zooko 三角形是一个与身份相关的概念, 要求命名系统在网络协议中具备安全性、去中心化等特征, 并对人类具有实际意义。通过推测可知, 系统仅可同时拥有两个这样的属性, 但是随着区块链以 Namecoin 的形式出现

后，这一问题也迎刃而解。然而，该方案并非万能，同时也面临着自身的挑战，比如用户是否能够以较为安全的方式存储和维护私钥。这就引出了去中心化的适用性问题。或许，去中心化方案并不适用于各种场合。一些较好的中心化系统在很多情况下都能运行得更好。



图 2.3 去中心化的生态系统概览

目前，许多项目均处于运作中，旨在向更为广泛的分布式区块链系统提供有效的解决方案。

随着去中心化这一类概念不断地被提及，一些新术语也频繁地出现于媒体和各种学术文献中。因此，在区块链背景下，有必要对此类新鲜事物予以解释。

2.5 智能合约

智能合约可视为一个去中心化的小型项目。智能合约并不一定需要区块链来运行。然而，考虑到区块链技术所提供的安全优势，可通过区块链作为智能合约的一个去中心化的执行平台，这一点基本上已形成了一种标准。智能合约通常包含一些业务逻辑和有限的数据库。相应地，区块链中的参与者可使用这些智能合约，或者代表网络参与者自主运行。

这些小程序驻留在区块链上，如果满足某些特定条件，即执行业务逻辑。智能合约的更多信息将在第 6 章加以介绍。

2.6 去中心化组织

去中心化组织（DO）表示在区块链上运行的软件程序，其思想源自人类社会中的各

种组织（涉及人和各项协议）。一旦 DO（以一种智能合约或一组智能合约的形式）被添加到区块链中，即呈现为去中心化特征，并且双方根据软件中定义的代码进行交互。

2.7 去中心化自治组织

就像 DO 一样，去中心化自治组织（DAO）也是一个计算机程序。该程序并不在区块链上运行，同时嵌入了管理和业务逻辑规则。DAO 和 DO 基本上是相同的，二者间的主要区别在于 DAO 具有自治性。这也意味着，执行过程完全自动化进行。DAO 包含了人工智能逻辑，而 DO 则缺少这一特性，且依赖于人工输入来执行业务逻辑。

在首次引入 DAO 时，Ethereum 区块链占主导地位。在 DAO 中，代码可视为管理实体，而不是人类或纸质合约。监管者作为自然人主体，负责代码的维护，以及社区提议的评估。如果从令牌持有者（参与者）处接收到足够的输入信息，DAO 即会“雇佣”来自外部的合约人。最著名的 DAO 项目是 The DAO (<https://daohub.org>)，该项目在众筹阶段即筹集了 1.68 亿美元。DAO 项目设计为一个风险投资基金，旨在提供一个去中心化的商业模式。另外，该项目中不存在任何单一实体作为持有者。然而，由于 DAO 代码中的一个错误，数百万美元的以太货币（ETH）从当前 DAO 转至由黑客创建的子 DAO 中。针对于此，需要在 Ethereum 区块链上使用一个硬分叉（hard fork），以挽回黑客所造成的影响，同时还须启动资金回收方案。这一事件引发了关于安全、质量和智能合约代码测试方面的争论，以确保实施完整性和有效的控制。目前正在进行的一些项目，特别是在学术界中，人们正在努力寻找智能合约编码的标准化形式。

目前，DAO 在司法界尚不存在法律地位，尽管 DAO 中可能会包含涉及编码协议和相关条件的智能代码，但这些规则在当前的现实世界法律体系中没有任何价值。或许有一天，由执法机构或监管机构委托和许可的自治主体可以嵌入到一个 DAO 中，以确保 DAO 在法律和法规意义上的完整性。这里，自治主体（AA）是一段无须人工干预的代码；而 DAO 则仅表示为去中心化的实体，并可在任意物理管辖范围内运行。因此，这里也提出了一个重要问题：当前的法律体系如何处理不同司法管辖区和地域间的 DAO？

2.8 去中心化自治企业

从概念上讲，DAC（去中心化企业）和 DAO 之间具有相似的含义，但常被认为是 DAO 的一个较小的子集。DAC 和 DAO 定义间有时会产生重叠，二者间的差别在于：DAO

通常具有非营利性质，而 DAC 可以通过向参与者提供股票或支付股息获得收益。企业可以在无须人工干预的情况下并根据内部编程逻辑自动运行业务。

2.9 去中心化自治社会

当前，分散的自治社会（DAS）还仅仅是一个概念。在大量复杂的智能合约的帮助下，社会整体都可以在区块链上运行，而 DAO 和去中心化的应用程序（DAPP）经结合后即可自动运行。这种模式并不意味着是一种非法行为，也不是一种完全自由主义的意识形态；相反，政府的许多服务可以通过区块链提供，如政府身份证系统、护照签发、契约、婚姻和生育记录系统。另一种理论是，如果一个政府腐败或中央系统不能提供社会所需要的、令人满意的信任级别，那么社会就可以在区块链上建立自己的虚拟社会，这是由去中心化的共识所驱动的，而且完全透明。这可能会被视为一种自由主义或半朋克式的梦想，但在区块链上是完全有可能的。

2.10 去中心化应用程序

之前提到的所有概念都需要在去中心化应用程序中加以实现。相应地，所有 DAO、DAC 和 DO 都可视作在对等网络中运行的、区块链之上的去中心化应用程序。这也体现了去中心化技术的一种进步。去中心化应用程序或 DAPP 可描述为：在自己的区块链上运行的软件程序；或者可使用另一个现有的区块链；抑或仅采用现有区块链解决方案中的相关协议，且分别被称作为 Type I 型、Type II 型和 Type III 型 DAPP。

2.10.1 去中心化应用程序的需求条件

去中心化应用程序须满足下列条件，这一定义源自 David Johnston 等人发表的白皮书 *The General Theory of Decentralized Applications, Dapps*。

- ❑ DAPP 应该是完全开源的，且不应存在任何实体可控制大多数的令牌。应用程序的所有更改都必须参考社区给出的反馈意见。
- ❑ 应用程序操作的数据和记录必须以加密方式实现安全保护，并存储在公共的、去中心化的区块链中，以避免出现任何中心故障。
- ❑ 应用程序必须使用一个加密令牌，以便为那些为应用程序贡献价值的用户提供访问和奖励，例如，比特币的矿工。

- ❑ 令牌必须由去中心化的应用程序根据标准的加密算法生成。该令牌作为对贡献者（例如，矿工）价值的证明。

2.10.2 DAPP 操作

通过使用共识算法，如工作量证明和权益证明等，即可实现基于 DAPP 的共识。到目前为止，仅 PoW 可抵制 51% 的攻击，这一点从比特币上可以明显看出。此外，DAPP 可以通过挖掘、筹款和开发来分发令牌（货币）。

下面考察与去中心化应用程序相关的一些示例。

1. KYC-Chain

该应用程序提供了一种方法，可管理 KYC 安全数据，同时也可视为一种基于智能合约的简便方法。

2. OpenBazaar

OpenBazaar 是一个去中心化的 P2P 网络，允许卖家和买家之间直接展开商业活动，而不是依赖于一个中心机构，这一点与 eBay 和亚马逊这一类传统的供应商有所不同。需要注意的是，该系统并不是建立在区块链之上的；相反，P2P 网络中采用了分布式哈希表，以便在对等点之间实现直接通信和数据共享。然而，OpenBazaar 利用比特币作为支付网络。

3. Lazooz

Lazooz 实际上是去中心化的 Uber，并支持 P2P 方式的拼车服务。用户可通过移动证明获得激励并赚取 Zooz 币。



注意：

除此之外，还存在大量的 DAPPS 并构建于 Ethereum 区块链上，读者可访问 <http://dapps.ethercasts.com/> 以获取更多信息。

2.11 去中心化平台

当前，存在大量去中心化的平台可供人们选择。与此相对应的是，世界各地的多家公司都推出了相关平台，并承诺分布式应用程序开发将变得更加容易、易于访问且兼具安全性，其中包括以下几个。

1. Ethereum

Ethereum 是第一个引入了图灵完备语言和虚拟机概念的区块链，这与比特币和许多其他加密货币中的脚本语言形成了鲜明的对比。随着图灵完备语言 Solidity 的出现，去中心化应用程序的开发旅程将具有无限的可能性。该语言是 Vitalik Buterin 于 2013 年提出的，同时还提供了一个公共区块链来开发智能合约和去中心化的应用程序。在 Ethereum 上，代币的名称为 Ethers。

2. Maidsafe

Maidsafe 提供了一个安全的（个人安全访问）网络，它由未使用的计算资源组成，如存储、处理能力和用户的数据连接。网络上的文件被划分为小块的数据，这些数据经加密后在整个网络中随机分布。另外，这一类数据只能由其各自的所有者进行检索。同时，网络将自动拒绝重复的文件，这可视为一个较为重要的创新之举，且有助于减少管理负载的额外的计算资源。Maidsafe 使用 Safecoin 作为代币来激励它的贡献者。

3. Lisk

Lisk 是一个区块链应用程序开发和加密货币平台。它允许开发人员使用 JavaScript 构建分散的应用程序，并将其托管在各自的侧链中。Lisk 使用委托证明（DPOS）机制来达成共识，即可以选出 101 个节点来保护网络并提出区块，并使用 Node.js 和 JavaScript 后端，而前端允许使用标准技术，如 CSS3、HTML5 和 JavaScript。Lisk 在区块链上使用 LSK 币作为货币。Lisk 的另一个衍生物是 Rise，这是一个基于 liska 的分散应用程序和数字货币平台。它更注重系统的安全性。

后续章节将对这一类平台以及其他平台加以具体讨论。

2.12 本章小结

本章介绍了去中心化的概念，这也是区块链技术提供的核心服务。虽然这一概念并非新鲜事物，但它在区块链中具有新的含义。因此，最近也产生了基于去中心化架构的各种应用。本章首先介绍了去中心化的基本概念。其次，讨论了区块链视角下的去中心化问题。此外，还介绍了区块链生态系统中不同层次的去中心化思想。随着区块链技术以及去中心化理念的出现，如 DAO、DAPP，一些新的概念和术语也应运而生。最后，本章还考察了一些去中心化应用实例。第 3 章将介绍区块链生态系统的基本概念，并重点考察加密技术，这也是区块链技术中重要的基础内容。

第3章 密码学和基本技术

本章主要讨论密码学中的概念、理论和实践内容，并结合区块链技术背景予以阐述。此外，还将讨论金融市场中的一些概念，以便为后续章节所涉及的内容提供理论基础。

除此之外，本章还将介绍加密算法的实际实现，以使读者实际体验加密函数的具体操作。为此，本章将使用到 OpenSSL 命令行。

在开始讨论理论基础之前，下面首先介绍 OpenSSL 的安装过程，以便读者在阅读理论性内容时展开某些实质性的工作。

Ubuntu Linux 发行版支持 OpenSSL，并通过以下命令进行安装：

```
$ sudo apt-get install openssl
```

下面首先阐述一些理论方面的基础内容，随后将展示一些与此相关的实际操作示例。

3.1 简介

密码学是保证信息安全的一门科学，并提供了一种安全的通信手段。其中，密码主要用于加密数据，因此，如果被对手截获且未被解密，数据对他们来说是没有任何意义的，因为解密过程需要使用到密钥。

密码学通常用于提供保密服务。就其本身而言，并不能视为一个完整的解决方案，而是大型安全系统中的一个重要构建模块，并以此解决安全问题。

密码学中包含了各种安全服务，如机密性、完整性、身份验证（实体身份验证和数据来源认证）和不可否认性。此外，在各种安全系统中还需要使用到问责制。

在进一步讨论密码学之前，需要首先解释一些数学术语和概念，以便完全理解本章后续内容。需要注意的是，本节主要对一些数学理论进行简要介绍，相关术语的证明和理论背景都将涉及严格的数学推导过程。当然，这也超出了本书的范围。关于这些主题的更多细节，读者可参考数论、代数或密码学等方面的书籍。

3.1.1 数学知识

密码学与数学紧密相关，本节将介绍一些基本概念，这些概念将有助于读者理解后续章节中的内容。

1. 集合

集合用于表述一组不同对象，例如， $X = \{1, 2, 3, 4, 5\}$ 。

2. 群

群定义为一个可交换集，其中，某个操作组合了集合中的两个元素。群操作处于闭合状态，并与定义的单位元关联。另外，集合中的每个元都存在一个逆元。闭包（闭合）意味着，如果元素 A 和 B 位于集合中，那么在元素执行操作之后的结果元素也位于集合中。这里，结合律（Associative）的意思是元素的分组不影响操作的结果。

3. 域

域表示为包含加法和乘法群的集合。更准确地说，集合中的所有元素都构成了一个加法和乘法群，同时满足加法和乘法的特定公理。所有的群运算也适用分配律。该定律规定，即使任何数据项或因子被重新排序，求和和乘积结果仍保持相同。

4. 有限域

有限域表示一个具有有限元素集的域，该结构也称作 Galois 域，这在密码学中特别重要——可以用来生成精确无误的数字运算结果。例如，在椭圆曲线密码学中使用有限素域来构造离散对数问题。

5. 阶数

阶数表示域中元素的数量，也称作域的基数。

6. 素域

素域是指包含素数元素的有限域，并对加法和乘法制定了专门的规则。另外，在素域里的每一个非零元素都有一个逆元。加法和乘法运算均采用模 p 进行计算。

7. 环

如果在一个阿贝尔群上可以定义多个操作，那么该群则演变为一个环。除此之外，还需满足其他一些属性。例如，环必包含有闭合性、结合性和分配性。

8. 循环群

循环群定义为一种群类型，并通过称作生成元的单一元素生成。换句话说，如果群操作重复应用于群中的某个特定元素，则可以生成群中的所有元素。

9. 阿贝尔群

当集合中元素的运算可交换时，即形成一个阿贝尔群。交换律基本上是指：改变元

素的顺序不会影响操作的结果, 例如, $A \times B = B \times A$ 。

10. 模运算

模运算也称作时钟运算, 当到达某一特定的数字时, 模运算中的数字将会以环绕方式出现。这一特定的数字定义称为模量的正数, 所有的操作都是在这—特定的数字上执行的。在时钟例子中, 存在从 1~12 的数字。当到达 12 时, 数字 1 再次出现。换句话说, 模运算是除法运算后的余数。例如, $50 \bmod 11$ 是 6, 因为 $50/11$ 的余数是 6。

上述内容简单地介绍了一些数学概念。下面将考察密码学。

3.1.2 密码学

如前所述, 密码学提供了各项安全服务, 其中涵盖了多项内容, 后续各小节将对相关概念逐一介绍。

3.1.3 保密性

保密性是指确保信息只对授权实体可用。

3.1.4 完整性

完整性是指确保信息只能由授权实体来修改。

3.1.5 认证

对于实体身份或消息的有效性, 验证过程提供了一种安全保证, 其中包括两种类型的认证方式。

1. 实体认证

实体认证确保实体参与某一通信会话并处于活动状态。传统意义上, 用户一般会发布用户名和密码, 用来访问正在使用的平台。这称作单因素认证, 因为该过程只存储一个因素, 即密码和用户名。考虑到各种原因 (如密码泄漏), 这种类型的身份验证并不安全。因此, 现在通常采用附加因素来提供更好的安全性保障。基于用户认证的附加技术称作多因素认证; 如果仅采用两种方法, 则称作双因素认证; 如果认证过程超出了两种因素, 则称作多因素认证。相关因素描述如下:

□ 第一种因素是用户所拥有的某些信息, 例如, 硬件标记或智能卡。在这种情况下

下,除了登录凭证之外,用户还可以使用硬件标记获得对系统的访问,这提供了两种身份验证因素的保护,即获取硬件标记以及登录凭证的用户将能够访问系统。这两种因素均可用于获得对系统的访问,因而该方法称作双因素身份验证机制。

- 第二种因素是使用生物特征来识别用户。在这种方法中,用户使用指纹、视网膜、虹膜或手掌几何特征为身份验证提供附加因素。通过这种方式,可以确保用户在身份验证机制中真实存在,因为生物特征对于个人来说是独一无二的。然而,为了确保高水平的安全性,仍需对此谨慎实施。一些研究表明,生物识别系统在某些情况下依然存在某些缺陷。

2. 数据源认证

数据源认证也称为消息身份验证,以确保对信息来源予以验证。数据源认证意味着数据的完整性,因为如果数据源被证实后,数据就不能被修改了。常见的各种方法包括消息认证码(MAC)和数字签名,这些术语将在后面的章节中详细解释。

3.1.6 不可否认性

不可否认性是指,通过不可伪造的证据,实体不可否定先前的承诺或行为。作为一项安全服务,这一机制提供了特定行为下的真实证据。在有争议的情况下,该属性是非常有必要的,此时,实体会拒绝执行操作,例如电子商务系统中的订单。该服务在电子交易中产生加密证据,以便在发生纠纷时可以作为行为确认。多年来,不可否认性一直是一个活跃的研究领域。电子交易中的纠纷是一个常见的问题,因而需要对此类问题加以重视,以提高服务消费者的可信度。

不可否认协议通常在通信网络中运行,并用于提供网络上某个实体(发起者或接收者)所用的行为证据。在这种情况下,有两种通信模型可以用来将消息从发起者 A 传递给接收者 B:

(1) 消息直接从发起者 A 发送到接收者 B。

(2) 消息发送至发起者 A 的代理,然后将消息传递给接收者 B。

不可否认协议主要涉及公平性、有效性和及时性。在许多场合下,某项事务往往会涉及多个参与者,而不是只有两个参与者。例如,在电子交易系统中,可以存在多个实体,例如清算代理、经纪人和交易员,并参与到单一的交易事务中。在这种情况下,双方不可否认协议将不再适用。为了解决这一问题,人们提出了多方不可否认协议。

3.1.7 问责制

问责制确保影响安全的行动可以追溯到责任方,通常由系统中的日志和审计机制提

供,因为在电子交易系统中,由于业务的性质,往往需要进行详细的审计。详细的日志对于跟踪一个实体的各项操作是至关重要的,例如,当某项交易置于基于日期和时间戳的审计记录中后,同时生产该实体的标识并保存在日志文件中。该日志文件可以选择加密,并作为数据库中的一部分内容,或者是系统上的独立 ASCII 文本日志文件。

3.2 密码原语

密码原语是安全协议或系统中的基本构建模块。后续小节将对加密算法予以介绍,这些算法对于构建安全协议和系统非常重要。这里,安全协议是指通过适当的安全机制,实现所需安全目标所采用的一系列步骤。

在实际应用中,存在多种安全协议,例如,认证协议、不可否认协议和密钥管理协议。

图 3.1 显示了一种通用的密码学模型。

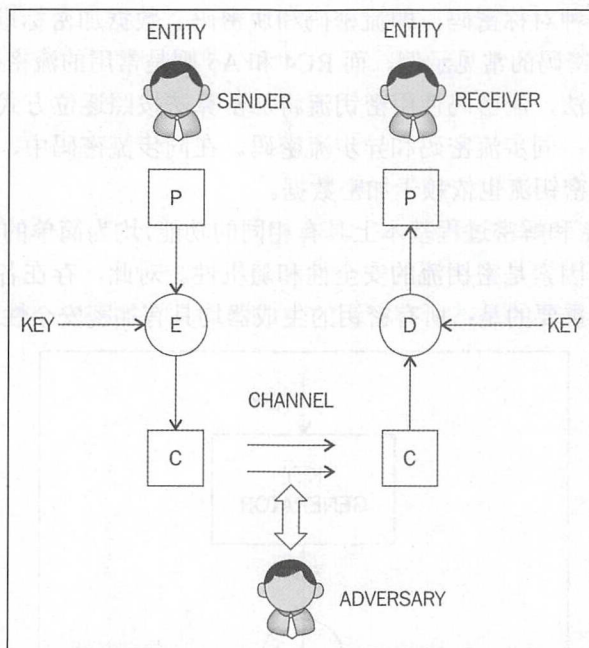


图 3.1 通用加密/解密模型

在图 3.1 中, P、E、C 和 D 分别表示纯文本、加密过程、密文和解密过程。基于当前所示的模型,相关概念包括实体、发送者、接收者、敌对者、密钥和通道,下面将对此逐一解释。

- ❑ 实体：指某人或某个系统，进而接收、发送或执行数据操作。
- ❑ 发送者：发送方是传输数据的实体。
- ❑ 接收者：接收方是接收数据的实体。
- ❑ 敌对者：尝试避开当前安全服务的实体。
- ❑ 密钥：密钥表示用于加密或解密数据的数据。
- ❑ 通道：提供了实体间的通信介质。

密码学主要分为两种类型，即对称加密和非对称加密。

3.2.1 对称加密

作为加密的一种类型，对称加密用来加密数据的密钥与解密数据是一样的，因此也称作共享密钥加密。在通信双方之间的数据交换之前，必须建立或同意密钥，这也是秘密密钥加密这一名称的由来。

相应地，存在两种对称密码，即流密码和块密码。数据加密标准（DES）和高级加密标准（AES）是块密码的常见示例，而 RC4 和 A5 则是常用的流密码。

作为一种加密算法，流密码使用密钥流将加密算法按照逐位方式应用到纯文本上。流密码包含两种类型：同步流密码和异步流密码。在同步流密码中，密钥流仅依赖于密钥，而异步流密码的密钥流也依赖于加密数据。

在流密码中，加密和解密过程基本上具有相同的功能，均为简单的模块 2 加法或 XOR 操作。流密码的关键因素是密钥流的安全性和随机性。对此，存在各种各样的技术可生成随机数，而且最为重要的是：所有密钥的生成器均具有加密安全性，如图 3.2 所示。

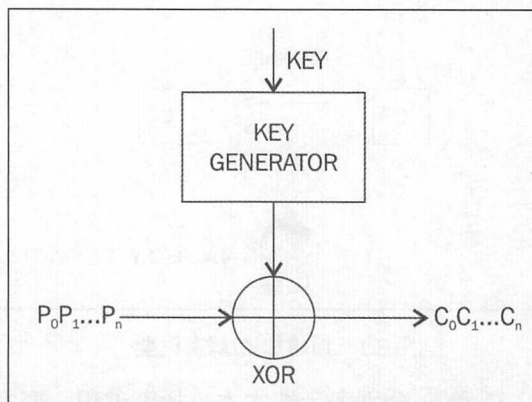


图 3.2 流密码操作

3.2.2 块密码

块加密算法将加密文本（明文）分为固定长度的块，并按照逐块的方式进行加密。块密码通常使用一种名为 Fiestel 密码的设计策略来构建。近期的块密码，如 AES (Rijndael)，则是采用替换和置换相结合的方式（称作替换-置换网络，SPN）进行构建的。

Fiestel 密码基于 Fiestel 网络，该结构首先由 Horst Fiestel 提出。这种结构整合了多轮重复性操作，以实现期望的加密属性，即混合和扩散。Fiestel 网络通过将数据分成两个块（左和右）进行操作，并通过密钥轮函数处理这些块。

混合操作使得加密文本与明文之间的关系变得复杂，在实际操作过程中，该操作是通过置换实现的。例如，纯文本中的“A”被加密文本中的“X”替换。在现代加密算法中，可使用名为 s - box 的查找表来执行替换。在加密的数据上，扩散属性对于加密数据采用了统计方式“散布”纯文本，以保证即使输入文本中的一个比特被改变，至少会改变密码文本中（平均）一半的位数。混合的作用主要体现在：找到加密密钥非常困难，即使许多加密和解密的数据对采用了相同的密钥创建。在实际操作过程中，混合行为通过移位或置换来实现。

Fiestel 密码的一个主要优点是加密和解密操作几乎相同，而且只需要对加密过程进行反向操作即可以实现解密。相应地，DES 是基于 Fiestel 密码的一个例子，如图 3.3 所示。

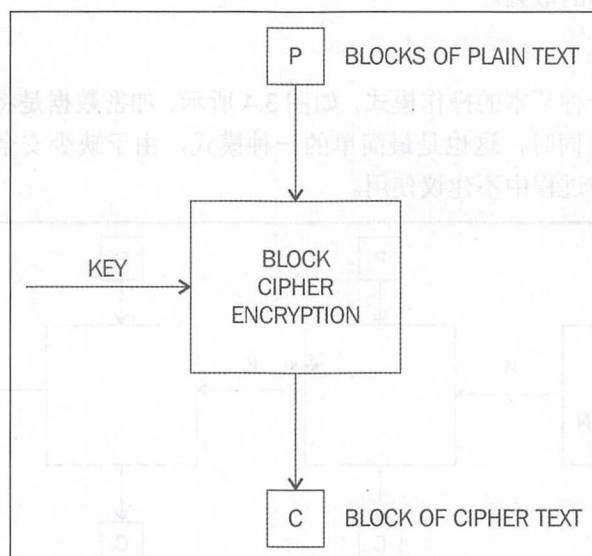


图 3.3 块密码的简化操作

块密码的各种操作方式包括电子代码书（ECB）、密码块链接（CBC）、输出反馈模式（OFB）或计数器模式（CTR）。这些模式用于指定加密函数与纯文本之间的应用方式，稍后将对此予以解释。

1. 块加密模式

在这种模式下，明文根据所使用的密码类型划分为固定长度块，然后在每个块上应用加密函数。

2. 密钥流生成模式

在这种模式下，加密函数生成一个密钥流，然后将其与纯文本流结合使用以实现加密。

3. 消息验证模式

在这种模式下，消息身份验证码计算为加密函数的结果。MAC 基本上是一个加密的校验和，并提供完整的服务。使用块密码生成 MAC 的最常见的方法是 CBC-MAC，在该方法中，该链的最后一个密码块部分用作 MAC。

4. 加密散列

散列函数主要用于将消息压缩为固定长度摘要。在这种模式下，块密码被用作压缩函数，以生成纯文本的散列。

5. 电子代码书

电子代码书是一种基本的操作模式，如图 3.4 所示。加密数据是将加密算法逐个应用于纯文本块的结果；同时，这也是最简单的一种模式，由于缺少安全性并且可以显示信息，因而在实际操作过程中不建议使用。

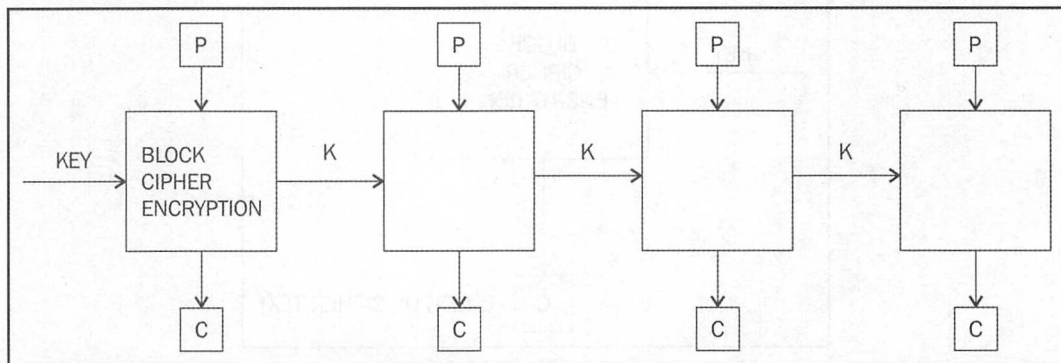


图 3.4 基于块密码的电子代码书模式

6. 密码块链接

在这种模式下，每一个纯文本块都与前面的加密块执行 XOR 操作。另外，CBC 模式使用初始化向量 IV 对第一个块进行加密。此处，建议随机选择 IV，如图 3.5 所示。

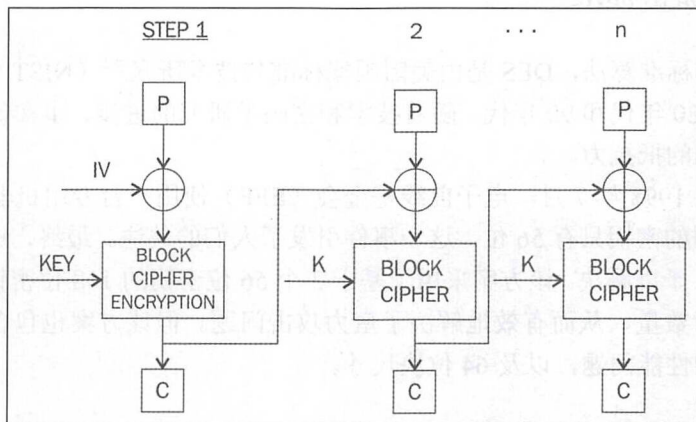


图 3.5 密码块链接模式

7. 计数器模式

CTR 模式采用块密码作为流密码。这种情况提供了唯一的 nonce 并将其与计数器值连接起来，从而生成一个密钥流，如图 3.6 所示。

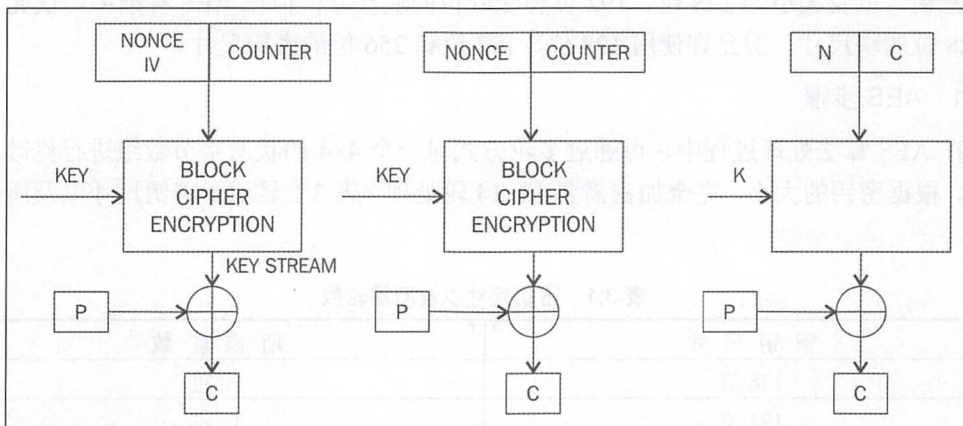


图 3.6 计数器模式

除此之外，还存在其他一些模式可用于多种场合，例如密码反馈模式 (CFB)、Galois 计数器模式 (GCM) 和输出反馈模式。

后续小节将讨论当前主流块密码的设计和 Related 机制，即 AES。针对 AES 标准，还将简要介绍一下数据加密标准（DES）的发展史。

3.2.3 数据加密标准

作为加密的标准算法，DES 是由美国国家标准与技术研究所（NIST）发布的，主要应用于 20 世纪 80 年代和 90 年代。随着技术和密码学研究的进步，事实证明，DES 对蛮力攻击缺少较强的抵抗力。

特别地，在 1998 年 7 月，电子前线基金会（EFF）使用一台专用机器破解了 DES。其中，DES 使用的密钥只有 56 位，这一事件引发了人们的关注。最终，这一问题通过三重 DES（3DES）予以解决。该方案采用了基于 3 个 56 位密钥的 168 位密钥，以及与 DES 算法相同的执行数量，从而有效地解决了蛮力攻击问题。但该方案也包含了某些限制条件，例如缓慢的性能问题，以及 64 位块尺寸。

3.2.4 高级加密标准（AES）

2001 年，在公开竞争之后，由密码学家 Joan Daemen 和 Vincent Rijmen 发明的一种名为 Rijndael 的加密算法被标准化为 AES，并在 2001 年由 NIST 进行了微小的修改。到目前为止，尚不存在比蛮力法更好的 AES 的攻击方法。最初的 Rijndael 算法允许使用不同的密钥，以及大小为 128 位、192 位和 256 位的块尺寸；但在 AES 标准中，仅允许使用 128 位的块尺寸。但允许使用 128 位、192 位和 256 位的密钥尺寸。

1. AES 步骤

在 AES 算法处理过程中，可通过多轮方式对一个 4×4 的状态字节数组进行修改。相应地，根据密钥的大小，完全加密需要 10~14 轮处理。表 3.1 显示了密钥尺寸以及所需的轮数。

表 3.1 密钥尺寸以及所需轮数

密 钥 尺 寸	所 需 轮 数
128 位	10 轮
192 位	12 轮
256 位	14 轮

一旦当前状态被初始化为密码输入内容，将在 4 个阶段中执行 4 项操作，进而对输入内容加密。对应各阶段分别为 AddRoundKey、SubBytes、ShiftRows 和 MixColumns。

(1) 在 AddRoundKey 步骤中, 状态数组与子密钥执行 XOR 运算, 该子密钥源自主密钥。

(2) 作为替换步骤, 使用一个查找表 (S 盒) 替换状态数组的所有字节。

(3) 除了状态数组中的第一个行, 该步骤以循环和递增的方式向左移动每一行。

(4) 在最后一个步骤中, 全部字节以线性方式按列混合。

上述各步骤描述了一轮 AES。在最后一轮 (视密钥大小而定, 轮数位 10、12 或 14) 中, 第 4 阶段通过 Addroundkey 替换, 以确保前 3 步无法简单地逆置, 如图 3.7 所示。

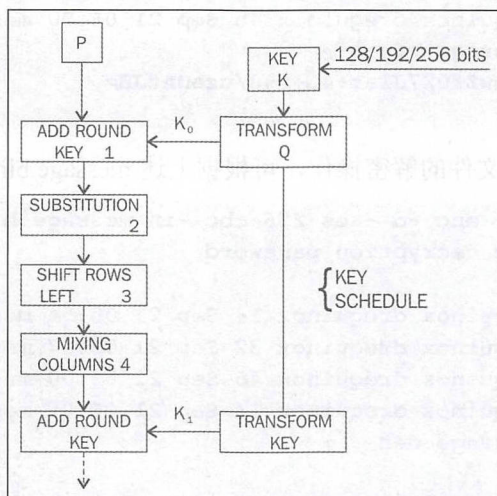


图 3.7 AES 块示意图, 并显示了第一轮处理。在最后一轮中, 未执行混合操作

各种加密货币电子钱包均使用了 AES 加密本地存储的数据。特别地, 在比特币电子钱包中, 使用了 CBC 模式的 AES 256。

2. 基于 AES 的加密/解密 OpenSSL 示例

在 OpenSSL 中, 基于 AES 的加密/解密过程如下所示:

```

~/Crypt$ openssl enc -aes-256-cbc -in message.txt -out message.bin
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
~/Crypt$ ls -ltr
total 12

-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
~/Crypt$ cat message.bin
  
```



```
Salted__w s_ŷ h~? :~/Crypt$
~/Crypt$
```

需要注意的是，message.bin 表示为二进制文件。出于兼容性和互操作性原因，可将该二进制文件编码为文本格式。对此，可采用下列命令：

```
~/Crypt$ openssl enc -base64 -in message.bin -out message.b64
~/Crypt$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
~/Crypt$ cat message.b64
U2FsdGVkX193uByIcwZf0Z7Jlat+4L+Fj8/uzEDAtJE=
~/Crypt$
```

为了实现加密 AES 文件的解密操作，可根据上述 message.bin 示例使用下列命令：

```
~/Crypt$ openssl enc -d -aes-256-cbc -in message.bin -out message.dec
enter aes-256-cbc decryption password:
~/Crypt$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 06:06 message.dec
~/Crypt$ cat message.dec
datatoencrypt
~/Crypt$
```

一些读者可能会注意到，即使全部块加密模式（除了 ECB 之外）均需要使用到初始化向量，但此处并未提供任何初始化向量，其原因在于：OpenSSL 自动从给定的密码中派生出初始化向量。用户可下列方式指定初始化向量：

```
-K/-iv, (Initialization Vector) should be provided in Hex
```

为了从 base64 中解码，可使用上一个示例中的 message.b64 文件，并输入下列命令：

```
~/Crypt$ openssl enc -d -base64 -in message.b64 -out message.ptx
~/Crypt$ ls -ltr
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 06:06 message.dec
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 06:16 message.ptx
~/Crypt$ cat message.ptx
Salted__w s_ŷ h~? :~/Crypt$
```

OpenSSL 中支持许多类型的密码,读者可以根据前面提供的示例考察相关选项。图 3.8 显示了所支持的密码类型列表。

```

Cipher Types
-aes-128-cbc             -aes-128-cbc-hmac-sha1   -aes-128-cbc-hmac-sha256
-aes-128-ccm             -aes-128-cfb             -aes-128-cfb1
-aes-128-cfb8            -aes-128-ctr             -aes-128-ecb
-aes-128-gcm             -aes-128-ofb             -aes-128-xts
-aes-192-cbc             -aes-192-ccm             -aes-192-cfb
-aes-192-cfb1            -aes-192-cfb8            -aes-192-ctr
-aes-192-ecb             -aes-192-gcm             -aes-192-ofb
-aes-256-cbc             -aes-256-cbc-hmac-sha1   -aes-256-cbc-hmac-sha256
-aes-256-ccm             -aes-256-cfb             -aes-256-cfb1
-aes-256-cfb8            -aes-256-ctr             -aes-256-ecb
-aes-256-gcm             -aes-256-ofb             -aes-256-xts
-aes128                  -aes192                  -aes256
-bf                       -bf-cbc                  -bf-cfb
-bf-ecb                  -bf-ofb                  -blowfish
-camellia-128-cbc         -camellia-128-cfb        -camellia-128-cfb1
-camellia-128-cfb8        -camellia-128-ecb        -camellia-128-cfb
-camellia-192-cbc         -camellia-192-cfb        -camellia-192-cfb1
-camellia-192-cfb8        -camellia-192-ecb        -camellia-192-ofb
-camellia-256-cbc         -camellia-256-cfb        -camellia-256-cfb1
-camellia-256-cfb8        -camellia-256-ecb        -camellia-256-ofb
-camellia128              -camellia192             -camellia256
-cast                     -cast-cbc                -cast5-cbc
-cast5-cfb               -cast5-ecb               -cast5-ofb
-des                      -des-cbc                 -des-cfb
-des-cfb1                 -des-cfb8                -des-ecb
-des-edc                  -des-edc-cbc             -des-edc-cfb
-des-edc-ofb              -des-edc3                -des-edc3-cbc
-des-edc3-cfb            -des-edc3-cfb1           -des-edc3-cfb8
-des-edc3-ofb            -des-ofb                  -des3
-desx                     -desx-cbc                -id-aes128-CCM
-id-aes128-GCM            -id-aes128-wrap          -id-aes192-CCM
-id-aes192-GCM            -id-aes192-wrap          -id-aes256-CCM
-id-aes256-GCM            -id-aes256-wrap          -id-smime-alg-CMS3DESwrap
-rc2                     -rc2-40-cbc              -rc2-64-cbc
-rc2-cbc                  -rc2-cfb                 -rc2-ecb
-rc2-ofb                  -rc4                      -rc4-40
-rc4-hmac-md5             -seed                     -seed-cbc
-seed-cfb                 -seed-ecb                 -seed-ofb
  
```

图 3.8 OpenSSL 中的选项

3.3 非对称加密

作为密码学的一种,非对称加密用于加密数据的密钥不同于解密数据的密钥,这也被称为公钥加密,并分别采用公钥和私钥加密和解密数据。当前应用包含了各种不对称加密方案,如 RSA、DSA 和 El-Gamall。

图 3.9 显示了公钥加密过程。

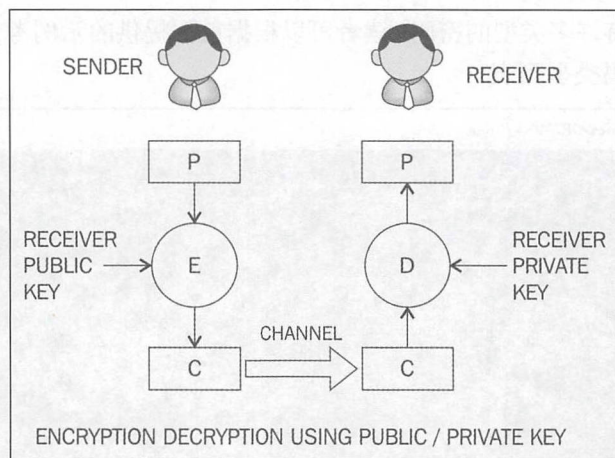


图 3.9 使用公钥/私钥的加密过程

图 3.9 解释了发送者如何使用接收者的公钥对数据进行加密,随后通过网络传输到接收者一方。一旦到达接收者处,就可以采用接收者的私钥解密。这样一来,私钥就保留在接收者的一方,因而不需要共享密钥来执行加密和解密过程,即对称加密。

图 3.10 显示了如何使用公钥加密来验证接收者接收消息的完整性。在该模型中,发送者使用私钥对数据进行签名,并将消息传递给接收者。一旦消息被接收方一端获取,就可以通过发送者的公钥验证其完整性。注意,该模型并没有进行加密操作,且仅用于消息验证和身份验证目的。

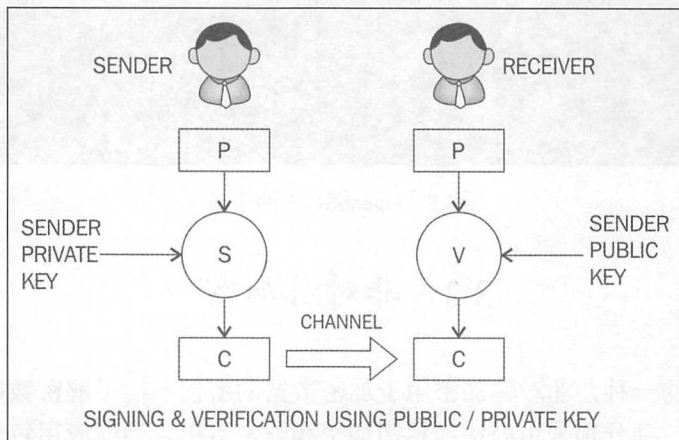


图 3.10 公钥加密签名模型

公钥密码机制提供的安全机制包括密钥建立、数字签名、验证、加密和解密。

密钥建立机制涉及协议的设计,并可在不安全通道上设置密钥。其中,不可否认服务是许多场合下非常理想的属性,可以通过数字签名来提供。某些时候,不仅要对用户进行身份验证,还要识别事务中涉及的实体,这也可以通过数字签名和挑战-应答协议的组合方式加以实现。最后,还可以使用公共密钥加密系统实现加密机制,如 RSA、ECC 或 El - Gammal。

与对称密钥算法相比,公钥算法的计算速度较慢。因此,一般不用于大型文件的加密或需要加密的只是数据。公钥算法通常用于交换对称算法的密钥,一旦密钥通过较为安全的方式建立,就可以使用对称密钥算法对数据进行加密。

公钥密码算法源自各种底层数学问题。下面介绍了不对称算法中的 3 种实现。

3.3.1 整数分解

此类方案源自如下事实:大整数一般难以分解。RSA 则是该算法类型的主要应用示例。

3.3.2 离散对数

该方案源自模运算中的一个问题:模函数易于计算,但无法计算生成器的指数。换言之,难以从计算结果中获取输入内容,这仅是一个单向函数。

例如,考察下列方程:

$$3^2 \bmod 10 = 9$$

当前问题可描述为:给定 9 后计算 2。相应地,生成器 3 的指数难以计算。这一困难问题常用于 Diffie-Hellman 密钥交换和数字签名算法。

3.3.3 椭圆曲线

基于前面讨论的离散对数问题。在椭圆曲线的上下文中,椭圆曲线表示为某一个场中的代数三次曲线,并可通过下列方程定义。该曲线具有非奇异特征,也就是说,不包含尖点和自相交结果。另外,该曲线包含两个变量 a 、 b 和一个无穷点。

$$y^2 = x^3 + ax + b$$

这里, a 、 b 表示为整数并可指定不同值,同时是椭圆曲线定义域中的元素。相应地,椭圆曲线可以定义为实数、有理数字、复数或有限域。针对加密功能,可采用素域上的椭圆曲线代替实数。此外,素数应该大于 3。通过改变 a 、 b 的值可以生成不同的曲线。

基于椭圆曲线的常用密码系统主要包括椭圆曲线数字签名算法 (ECDSA) 和椭圆曲线 Diffie-Hellman (ECDH) 密钥交换算法。

3.4 公钥和私钥

针对公钥加密，首先须考察公钥和私钥这两个概念。

顾名思义，私钥基本上是一个随机生成的数字且兼具保密性，并被用户私下持有。另外，私钥应受到保护，并且不应授予该密钥未授权的访问；否则，整个公钥加密方案将受到危害——该密钥用于解密消息。私钥的长度取决于所使用的算法的类型和分类。例如，在 RSA 算法中，通常使用 1024 位或 2048 位的密钥。1024 位的密钥缺少应有的安全性，在实际操作过程中建议至少使用 2048 位。

公钥是公私-密钥对的公共部分，公钥可公开获取并由私钥持有者发布。若希望对公钥发布者发送加密消息，可使用已发布的公钥对消息进行加密，并将其发送到私钥的持有者。由于相应的私钥被指定的接收者安全地持有，因而消息将无法被解密。一旦公钥加密消息被接收，接收者就可以使用私钥解密消息。对于公钥，还需关注其他一些问题，例如公钥发布者的真实性和识别过程。

3.4.1 RSA

RSA 于 1977 年由 Ron Rivest、Adi Shamir 和 Leonard Adelman 发明，因此得名 RSA。RSA 基于整数分解问题，两个大素数的乘法运算易于实现，但很难将其分解为两个原始数字。

RSA 算法的核心内容位于密钥的生成过程中。RSA 密钥对的生成过程包含下列步骤：

(1) 生成模数。

□ 选取 p 和 q 为大素数。

□ p 和 q 执行乘法运算，生成模数 $n=p.q$ 。

(2) 生成互质数。

□ 定义数值 e 。

□ e 应该满足一定条件，即大于 1 且小于 $(p-1)(q-1)$ 。换句话说，除 1 以外的任何数字都可以被分成 e 和 $(p-1)(q-1)$ ，即互质数。也就是说， e 表示 $(p-1)(q-1)$ 的互质数。

(3) 生成公钥。

步骤 (1) 中生成的模数和步骤 (2) 中生成的 e 定义为一对公钥，表示为公共部分内容并可与任何人分享；然而， p 和 q 则具有保密性。

(4) 生成私钥。

私钥表示为 d ，并通过 p 、 q 和 e 进行计算。基本上，私钥定义为 e 模 $(p-1)(q-1)$ 的逆，

对应的方程形式如下所示：

$$ed = 1 \bmod (p-1)(q-1)$$

通常情况下，可采用扩展后的欧几里德算法计算 d 。该算法通过 p 、 q 和 e 计算 d 。该方法核心理念可描述为：若 p 和 q 已知，通过扩展的欧几里德算法即可方便地计算密钥 d 。这也表明，对于模数 n 而言， p 和 q 应足够大，以使分解过程难以实施（无法计算）。

1. 基于 RSA 的加密/解密操作

RSA 使用下列方程生成密文：

$$C = P^e \bmod n$$

这意味着将纯文本 P 升至 e 次数，然后将其减至模数 n 。RSA 的解密由以下等式给出：

$$P = C^d \bmod n$$

这表明，如果接收方持有公钥对 (n, e) ，可将 C 升至私钥 d 值，并减至模数 n 来解密数据。

2. 椭圆曲线加密 (ECC)

ECC 基于离散对数问题，以及有限域 (Galois 域) 上的椭圆曲线。与其他公钥算法类型相比 (例如 RSA)，ECC 的优势主要体现在：较小的密钥尺寸即可提供相同的安全级别。另外，源自 ECC 的两种较为著名的方案是：针对密钥交换的椭圆曲线 Diffie-Hellman，以及针对数字签名的椭圆曲线数字签名算法 (ECDSA)。除此之外，ECC 还可用于加密操作，但在实际操作过程中却较少使用，而是常见于密钥交换和数字签名中。鉴于 ECC 仅需较少的操作空间，因而在嵌入式平台中或者存储资源较为紧张的系统较为流行。与 RSA 中的 3072 个字节相比，对于相同的安全级别，ECC 仅需采用 256 位的操作数即可。

为了进一步理解 ECC，下面简要地介绍一下该问题背后的数学知识。基本上讲，椭圆曲线表示为某种类型的多项式，即 weierstrass 方程，并在有限域中生成一条曲线。其中，最为常用的域可描述为：全部数学运算仅涉及模数 a 和素数 p 。椭圆曲线组由有限域上的曲线点构成。

相应地，椭圆曲线方程如下所示：

$$y^2 = x^3 + Ax + B \bmod p$$

其中， A 和 B 属于有限域 Z_p 或 FP (素域)，以及一个称为无穷远点的特殊值。无穷远点 ∞ 用于为曲线点提供同一性操作。

此外，还需要满足另一个条件，以确保前述方程不包含重复根。这意味着曲线是非奇异的。



更准确地说，作为标准条件，这确保了曲线的非奇异性，如下所示：

$$4a^3 + 27b^2 \neq 0 \pmod{p}$$

图 3.11 显示了椭圆曲线的实数表达方式。相应地，实数方程如下所示：

$$y^2 = x^3 + ax + b$$

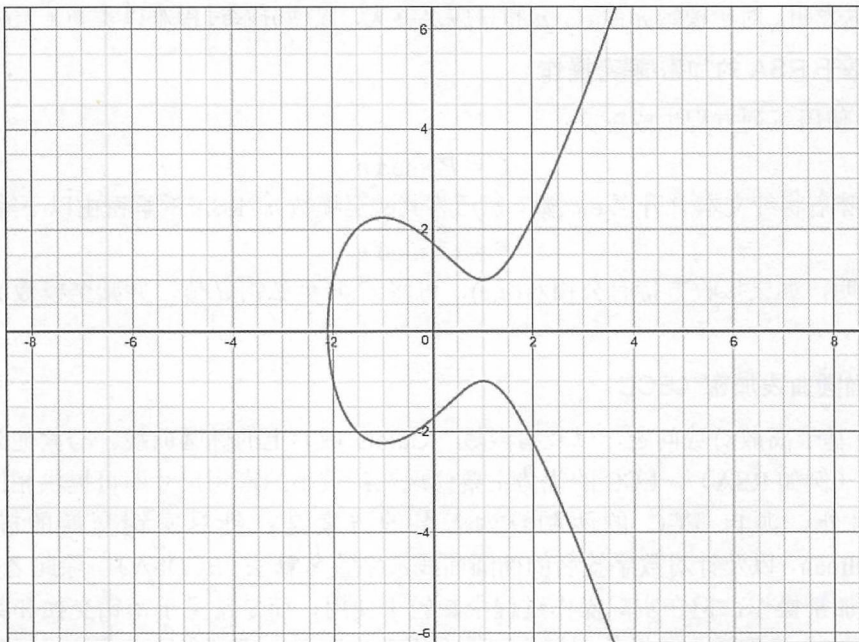


图 3.11 实数上的椭圆曲线。其中， $a = -3$ ， $b = 3$

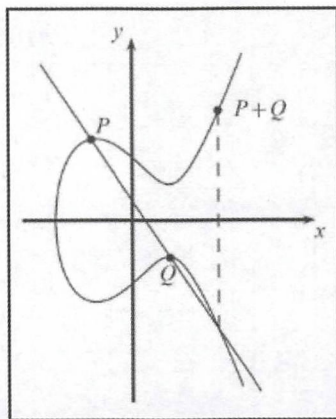
用于椭圆曲线加密时的实际曲线位于有限素域，但此处显示为实数，其原因在于当在 R 上显示图像时，可视化过程将变得更加简单。

当构建基于椭圆曲线的离散对数问题时，需要定义一个足够大的循环群。首先，可将群元素标识为一组点（满足之前所讨论的方程）。随后，需要在这些点上定义群操作。

3. 点加法

椭圆曲线上的群运算包括点加法和倍点（point doubling）计算。点加法是指两个不同的点相加；而倍点计算表示同一个点之间的自身相加。两种操作都可以实现相应的可视化结果。

图 3.12 显示了点加法运算，即椭圆曲线上点加法运算的几何表达方式。在该方法中，直线穿越当前曲线，并与曲线相交于 P 、 Q 两点。相应地，这将生成曲线和直线之间的第 3 个点，并镜像为 $P+Q$ ，即相加结果 R 。

图 3.12 R 上的点加法可视化结果

由符号“+”表示的群操作将生成下列等式：

$$P+Q=R$$

在当前示例中，两个点相加可计算曲线上第3个点的坐标，如下所示：

$$P+Q=R$$

更为准确地讲，坐标将按照下列方式相加：

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

点加法等式如下：

$$x_3 = s^2 - x_1 - x_2 \mod p$$

$$y_3 = s(x_1 - x_3) - y_1 \mod p$$

最终结果如下：

$$S = \frac{(y_2 - y_1)}{(x_2 - x_1)} \mod p$$

其中， S 表示穿越 P 和 Q 的一条直线。

这里展示的点加法示例是使用 Certicom 在线计算器生成的，表示有限域 F_{23} 上的加法和方程解。这与前面所展示的例子形成了鲜明的对比——位于实数域上，只显示了曲线，但不包含方程的解，如图 3.13 所示。

在当前示例中，图中左侧显示了满足下列等式的点：

$$y^2 = x^3 + 7x + 11$$

在有限域 F_{23} 中，上述方程存在 27 个解。其中，可选取 P 和 Q 生成点 R 。相应地，计算结果显示在图中右侧，并计算第 3 个点 R 。注意，此处 l 被用来描述通过 P 和 Q 的直线。

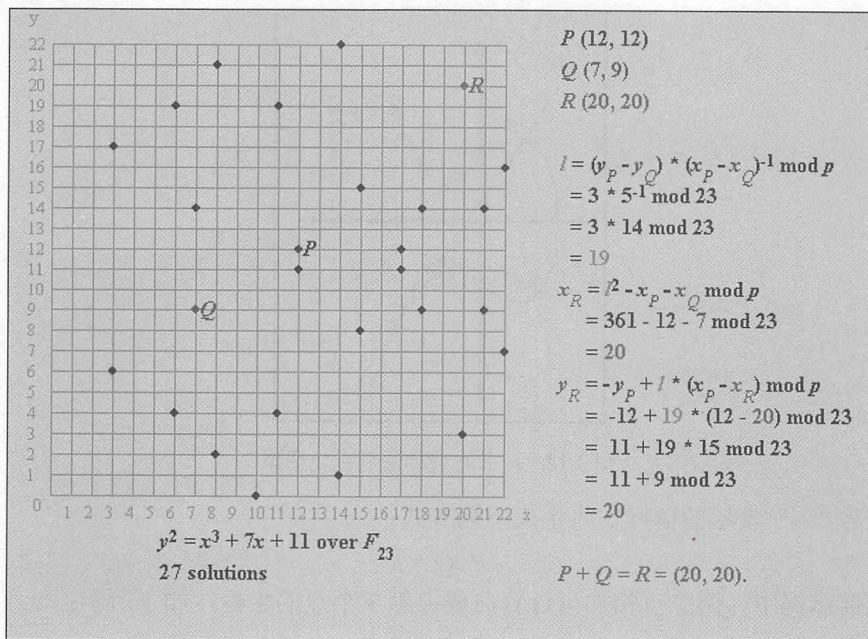


图 3.13 使用 Certicom 在线计算器生成的点加法示例

关于方程与图中点之间的匹配方式，作为示例，此处选取了 $x=3$ 和 $y=6$ 处选取了点 (x,y) 。最后，相关值满足当前方程，如下所示：

$$y^2 \bmod 23 = x^3 + 7x + 11 \bmod 23$$

$$6^2 \bmod 23 = 3^3 + 7(3) + 11 \bmod 23$$

$$36 \bmod 23 = 59 \bmod 23$$

$$13 = 13$$

下面考察倍点概念，这也是可在椭圆曲线执行的另一项操作。

4. 倍点

倍点是椭圆曲线上的另一种群运算，如图 3.14 所示，这是一个 P 自身相加的过程。在这种方法中，沿当前曲线绘制一条切线。在切线和曲线的交点处将得到第二个点。随后，该点将被镜像，并生成最终结果 $2P = P + P$ 。

在倍点运算中，对应方程如下所示：

$$x_3 = s^2 - x_1 - x_2 \bmod p$$

$$y_3 = s(x_1 - x_3) - y_1 \bmod p$$

$$S = \frac{(y_2 - y_1)}{(x_2 - x_1)} \bmod p$$

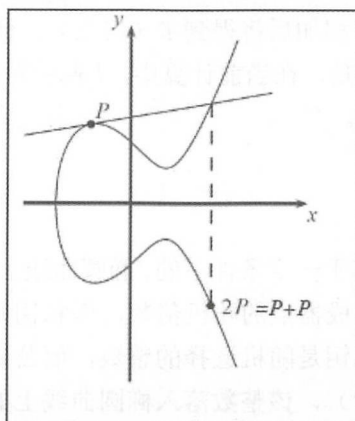


图 3.14 实数域上的倍点运算

此处, S 表示为途经 P 的切线斜率, 即图 3.14 中上方的直线。在上一个例子中, 曲线仅表示为实数点, 且未涉及方程解。

下面的示例显示了有限域 F_{23} 中椭圆曲线的解和倍点。在图 3.15 中, 左侧内容显示了满足当前方程的各个点。

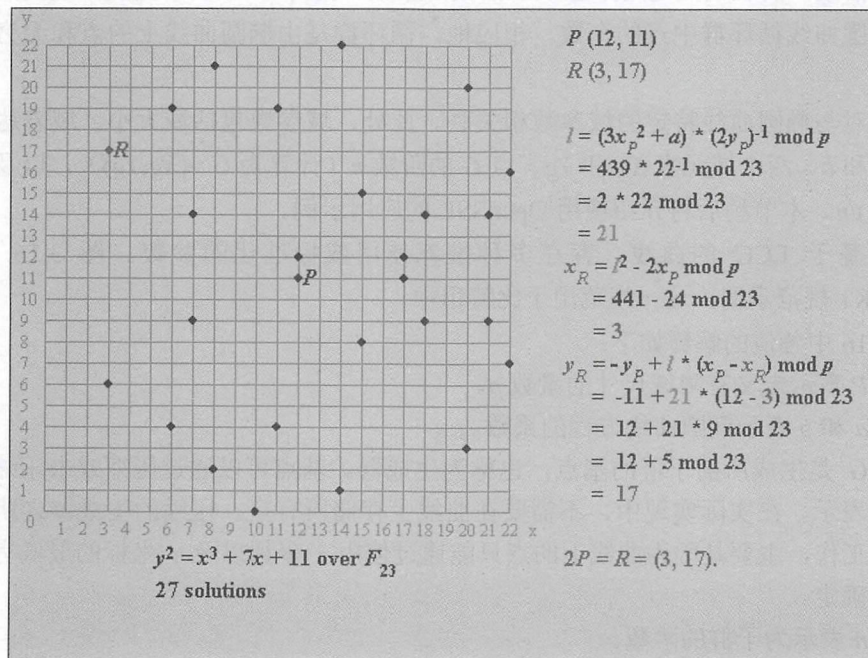


图 3.15 使用 Certicom 在线计算器实现的倍点



图中右侧内容表示 P 自身相加后将得到 R (倍点)。此处不存在 Q ，同一点 P 将用于计算加倍过程。需要注意的是，在当前计算中， l 表示为途经 P 的切线。

下面将讨论离散对数问题。

3.4.2 离散对数问题

ECC 的离散对数问题是基于一定条件下的，椭圆曲线上的所有点可组成一个循环群。

在椭圆曲线上，公钥是生成器点的随机倍数，而私钥则是一个随机选择的、用于生成倍数的整数。换句话说，私钥是随机选择的整数，而公钥则是曲线上的一个点。离散对数问题用于计算私钥 (整数)，该整数落入椭圆曲线上的所有点上。稍后将通过方程形式对此予以准确描述。

考察一条椭圆曲线 E ，其中包含了两个元素 P 和 T 。离散对数问题将计算整数 d ，其中 $1 \leq d \leq \#E$ 且有

$$P + P + \cdots + P = dP = T$$

此处， T 表示为公钥 (曲线上的点)， d 则表示为私钥。换句话说，公钥定义为生成器的随机倍数，而私钥则是用来生成该倍数的整数。另外， $\#E$ 表示椭圆曲线的阶数，主要是指椭圆曲线循环群中点的个数。相应地，循环群是由椭圆曲线上的点和无穷处的点组成的。

密钥对与椭圆曲线特定的域参数相关联。此处，域参数包括域大小、域表达方式、域元素 a 和 b 、两个域元素 X_g 和 Y_g 、点 G 的阶数 n (计算为 $G=(X_g, Y_g)$)，以及辅因子 $h = \#E(F_q)/n$ 。本节稍后将介绍使用 OpenSSL 的应用示例。

针对基于 ECC 的曲线，存在多种推荐使用或标准化的参数。图 3.16 显示了 SECP256K1 规范示例，这一规范用于比特币中。

图 3.16 中各值的解释如下：

- ❑ P 表示定义有限域尺寸的素数 p 。
- ❑ a 和 b 表示椭圆曲线方程的系数。
- ❑ G 是生成所需子群的基点，也称为生成器。基点可以通过压缩或未压缩的形式表示。在实际实现中，不需要在曲线上存储所有点。压缩的生成器之所以能够工作，主要是因为曲线上的点只能通过使用 x 坐标以及 y 坐标的最低有效位来确定。
- ❑ n 表示为子群的阶数。
- ❑ h 表示子群的辅因子。



The elliptic curve domain parameters over \mathbb{F}_p associated with a Koblitz curve secp256k1 are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field \mathbb{F}_p is defined by:

```
p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
    FFFFFFFF
    =  $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ 
```

The curve $E: y^2 = x^3 + ax + b$ over \mathbb{F}_p is defined by:

```
a = 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000
b = 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000007
```

The base point G in compressed form is:

```
G = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9
    59F2815B 16F81798
```

and in uncompressed form is:

```
G = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9
    59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448
    A6855419 9C47D08F FB10D4B8
```

Finally the order n of G and the cofactor are:

```
n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF BAAEDCE6 AF48A03B BFD25E8C
    D0364141
h = 01
```

图 3.16 SECP256K1 规范示例 (<http://www.secg.org/sec2-v2.pdf>)

下面将通过 OpenSSL 的一个示例来帮助读者了解 RSA 的实际应用,并讨论如何采用 OpenSSL 生成 RSA 公钥和私钥对。

1. 公钥和私钥对的生成方式

下面首先介绍基于 OpenSSL 的 RSA 私钥生成方式,如下所示:

```
~/Crypt$ openssl genpkey -algorithm RSA -out privatekey.pem -pkeyopt
rsa_keygen_bits:1024
.....++++++
.....++++++
```

待执行了上述命令后,将生成名为 privatekey.pem 的文件,其中包含了所生成的密钥,如下所示:

```
~/Crypt$ cat privatekey.pem
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAKJOFBzPy2vOd6em
Bk/UGrzdY7TvgDYnYxBfiEJId/r+EyMt/F14k2fDToVwxXaXTxiQgD+BKuiey/69
9itnrqW/xy/pocDMvobj8QCngEntOdNoVSa+n+t0f9nRM3iVM94mz3/C/v4vXvoac
```




```
PyPkr/0jhIV0woCurXGTghgqIbHRAgMBAAECgYEA1B3s/N41Jh011TkOSYunWtzT
6isnNkR7g1WrY9H+rG9xx4kP5b1DyE3SvxBLJA6xgBle8JVQMzm3sKJrJPFZzzT5
NNNNugCxaixcF1mPzJAP3aqpcSjxKpTv4qqqYevwgW1A0R3xKQZzBKU+bTO2hXV
D1oHxu75mDY3xCwqSAECQQDUYV04wNSEjEy9tYJ0zaryDAcvd/VG2/U/6qiQGajB
eSpSgoEESigbusKku+wVtRYgWWEomL/X58t+K01eMMZzAkeAw6PUR9YLebsm/Sji
iOShV4AKuFdi7t7DYWE5U1bluqP/i28zN/ytt4BXKIs/KcFykQGeAC6LDHZyycyc
ntDIOQJAVqrE1/wYvV5jkqcXbYLgV5YA+KYDOb9Y/ZRM5UETVKCVXNanf5Cjfw1h
MMhfNxyGwvy2YVK0Nu8oY3xYPi+5QQJAUGcmORe4w6Cs12JUJ5p+zG0s+rG/URhw
B7djTXm7p6b6wR1EWYAZDM9MArenj8uXAA1AGCcIsmiDqHfU7lgz0QJAe9mOdNGW
7qRppgmOE5nuEbxbDSQI7OqHYbOLuwfCjHzJBRsgqyi6pj9/9CbXJrZPgNDwdLEb
GgpDKtZs9gLv3A==
-----END PRIVATE KEY-----
```

由于私钥在数学上与公钥有所关联，因而可以从私钥中生成或派生公钥。根据上述私钥示例即可生成公钥，如下所示：

```
~/Crypt$ openssl rsa -pubout -in privatekey.pem -out publickey.pem
writing RSA key
```

公钥可采用文本阅读器或文本查看器进行查看，如下所示：

```
~/Crypt$ cat publickey.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiTThQcz8trzenpgZP1Bq8w8u0
74A2J2MQX4hCSHF6/hMjLfxdeJNnw0zlcMV2108YkIA/gSronsv+vfYrZ661v8cv
6aHAzL6G4/EAp4BJ7TnTaFUmjfrdH/Z0TN41TPeJs9/wv7+L176GnD8j5K/9I4SF
dMKArq1xk4IYKiGx0QIDAQAB
-----END PUBLIC KEY-----
```

关于各种组件的更多细节内容，例如模数、过程中使用的素数、生成私钥的指数和系数，可以使用下面的命令（完整的输出将占用较大的篇幅）：

```
~/Crypt$ openssl rsa -text -in privatekey.pem
Private-Key: (1024 bit)
modulus:
    00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
    1a:bc:c3:cb:b4:ef:80:36:27:63:10:5f:88:42:48:
    77:fa:fe:13:23:2d:fc:5d:78:93:67:c3:4c:e5:70:
    c5:76:97:4f:18:90:80:3f:81:2a:e8:9e:cb:fe:bd:
    f6:2b:67:ae:a5:bf:c7:2f:e9:a1:c0:cc:be:86:e3:
    f1:00:a7:80:49:ed:39:d3:68:55:26:8d:fa:dd:1f:
    f6:74:4c:de:25:4c:f7:89:b3:df:f0:bf:bf:8b:d7:
    be:86:9c:3f:23:e4:af:fd:23:84:85:74:c2:80:ae:
    ad:71:93:82:18:2a:21:b1:d1
```



```

publicExponent: 65537 (0x10001)
privateExponent:
    00:94:1d:ec:fc:de:25:26:1d:25:d5:39:0e:49:8b:
    a7:5a:dc:d3:ea:2b:27:36:44:7b:83:55:ab:63:d1:
    fe:ac:6f:71:c7:89:0f:e5:bd:43:c8:4d:d2:bf:10:
    4b:24:0e:b1:80:19:5e:f0:95:50:33:39:b7:b0:a2:
    6b:24:f1:59:cf:34:f9:34:d3:67:ba:00:b1:6a:2a:
    f1:70:5d:66:3f:32:40:3f:76:aa:a5:c4:a3:c4:aa:
    53:bf:8a:a0:a9:87:af:c2:05:b5:03:44:77:c4:a4:
    19:cc:12:94:f9:b4:ce:da:15:d5:0f:5a:07:c6:ee:
    f9:98:36:37:c4:2c:2a:48:01
prime1:
    00:d4:61:5d:38:c0:d4:84:8c:4c:bd:b5:82:74:cd:
    aa:f2:0c:07:2f:77:f5:46:db:f5:3f:ea:a8:90:19:
    a8:c1:79:2a:52:aa:81:04:4a:28:1b:ba:c2:a4:bb:
    ec:15:b5:16:20:59:61:28:98:bf:d7:e7:cb:7e:2b:
    4d:5e:30:c6:59
prime2:
    00:c3:a3:d4:47:d6:0b:79:bb:26:fd:28:e2:88:e4:
    a1:57:80:0a:b8:57:62:ee:de:c3:61:61:39:52:56:
    f5:ba:a3:ff:8b:6f:33:37:fc:ad:b7:80:57:28:8b:
    3f:29:c1:72:91:01:9e:00:2e:8b:0c:76:72:c9:cc:
    9c:9e:d0:c8:39

```

类似地，可以使用以下命令来查看公钥。其中，公钥和私钥均为 base64 编码。

```

~/Crypt$ openssl pkey -in publickey.pem -pubin -text
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiT hQcz8trznepngZP1Bq8w8u0
74A2J2MQX4hCSHF6/hMjLfxdeJNnw0zlcMV2108YkIA/gSrons v+vfYrZ66lv8cv
6aHAzL6G4/EAp4BJ7TnTaFUmjfrdH/Z0TN41TPeJs9/wv7+L176GnD8j5K/9I4SF
dMKArq1xk4IYKiGx0QIDAQAB
-----END PUBLIC KEY-----
Public-Key: (1024 bit)
Modulus:
    00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
    1a:bc:c3:cb:b4:ef:80:36:27:63:10:5f:88:42:48:
    77:fa:fe:13:23:2d:fc:5d:78:93:67:c3:4c:e5:70:
    c5:76:97:4f:18:90:80:3f:81:2a:e8:9e:cb:fe:bd:
    f6:2b:67:ae:a5:bf:c7:2f:e9:a1:c0:cc:be:86:e3:
    f1:00:a7:80:49:ed:39:d3:68:55:26:8d:fa:dd:1f:
    f6:74:4c:de:25:4c:f7:89:b3:df:f0:bf:bf:8b:d7:
    be:86:9c:3f:23:e4:af:fd:23:84:85:74:c2:80:ae:

```




```
ad:71:93:82:18:2a:21:b1:d1
Exponent: 65537 (0x10001)
```

现在，公钥可以公开共享，任何想给我们发送消息的人都可以使用公钥加密消息，并发送给我们。然后，我们可以使用相应的私钥对文件进行解密。

2. 如何采用 OpenSSL 和 RSA 加密和解密

首先，用第一个示例演示如何使用 RSA 执行加密操作。

根据之前生成的私钥，可对文本文件 message.txt 加密，对应命令如下所示：

```
:~/Crypt$ openssl rsautl -encrypt -inkey publickey.pem -pubin -in
message.txt -out message.rsa
```

这将生成名为 message.rsa 的文件，该文件为二进制格式。如果在 nano 编辑器中打开 message.rsa 文件，结果如图 3.17 所示。

```
drequinox@drequinox-OP7010: ~/Crypt
GNU nano 2.4.2      File: message.rsa

^@o9
^@^@bba^@8 ^@^@^@^@^@-8 ^@E#^@I^@X$uxM^@3^@Lx{k^@P^@^@>^@Cv^@v^@^@^@^@^@jA#^@^@R^@^@j^@e^@G
^@E7^@2^@k^@k^@^@Q^@^@F^@k^@k^@U^@^@~^@
-^@^@NkV^@^@RoO~^@D^@p^@i^@^@2^@CmU^@C^@^@+a^@^@F^@M'^@B^@A^@^@

^G Get Help      ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify      ^C Cur Pos
^X Exit          ^R Read File    ^\ Replace      ^U Uncut Text   ^T To Spell     ^_ Go To Line
```

图 3.17 message.rsa 文件显示的乱码

当解密 RSA 加密文件时，可使用下列命令：

```
:~/Crypt$ openssl rsautl -decrypt -inkey privatekey.pem -in message.rsa -
out message.dec
```

当前，如果使用 cat 命令阅读该文件，则可显示解密后的纯文本内容，如下所示：

```
:~/Crypt$ cat message.dec
datatoencrypt
```

3. 基于 OpenSSL 的 ECC

OpenSSL 提供了异常丰富的函数库，进而可执行椭圆曲线加密操作。下面将从应用角度考察 OpenSSL 中的 ECC 函数。

❑ ECC 私钥和公钥对

下面首先展示使用 OpenSSL 库中的 ECC 函数创建私钥。

❑ 私钥

ECC 基于不同标准所定义的域函数。针对 OpenSSL 中现有标准所定义和推荐的曲线，当使用下列命令时，即可查看 OpenSSL 中的全部列表：

```
Crypt$ openssl ecparam -list_curves
secp112r1 : SECG/WTLS curve over a 112 bit prime field
secp112r2 : SECG curve over a 112 bit prime field
secp128r1 : SECG curve over a 128 bit prime field
secp128r2 : SECG curve over a 128 bit prime field
secp160k1 : SECG curve over a 160 bit prime field
secp160r1 : SECG curve over a 160 bit prime field
secp160r2 : SECG/WTLS curve over a 160 bit prime field
secp192k1 : SECG curve over a 192 bit prime field
secp224k1 : SECG curve over a 224 bit prime field
secp224r1 : NIST/SECG curve over a 224 bit prime field
secp256k1 : SECG curve over a 256 bit prime field
secp384r1 : NIST/SECG curve over a 384 bit prime field
secp521r1 : NIST/SECG curve over a 521 bit prime field
prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field
.
.
.
.
brainpoolP384r1: RFC 5639 curve over a 384 bit prime field
brainpoolP384t1: RFC 5639 curve over a 384 bit prime field
brainpoolP512r1: RFC 5639 curve over a 512 bit prime field
brainpoolP512t1: RFC 5639 curve over a 512 bit prime field
```

由于完整的输出结果将占用较大的篇幅，因而此处仅展示了部分内容。在下面的示例中，将采用 SECP256k1 解释 ECC 的应用方式。

❑ 生成私钥

相关文件如下所示：

```
~/Crypt$ openssl ecparam -name secp256k1 -genkey -noout -out ecprivatekey.pem
~/Crypt$ cat ec-privatekey.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIjHUIm9NZAgfpUrSxUk/iINqlghM/ewn/RLNreur52h/oAcGBSuBBAK
oUQDQgAE0G33mCZ4PKbg5EtwQjk6ucv9Qc9DTr8JdcGXYGxHdzr0Jt1NInaYE0GG
ChFMT5pK+wfvSLkYl5ul0oczwWKjng==
-----END EC PRIVATE KEY-----
```

上述文件名为 ec-privatekey.pem, 其中包含了根据 SECP256K1 曲线生成的 EC 私钥。当从私钥中生成公钥时, 可输入下列命令:

```
~/Crypt$ openssl ec -in ec-privatekey.pem -pubout -out ec-pubkey.pem
read EC key
writing EC key
```

读取该文件将可生成以下输出结果, 并显示创建完毕的公钥:

```
~/Crypt$ cat ec-pubkey.pem
-----BEGIN PUBLIC KEY-----
MFYwEAYHkoZIZj0CAQYFK4EEAAoDQgAE0G33mCZ4PKbg5EtwQjk6ucv9Qc9DTr8J
dcGXYGxHdzr0Jt1NInaYE0GGChFMT5pK+wfVSLkYl5ul0oczWkIng==
-----END PUBLIC KEY-----
```

当前, ec-pubkey.pem 文件涵盖了源自 ec-privatekey.pem 文件的公钥。通过下列命令, 还可进一步查看私钥:

```
~/Crypt$ openssl ec -in ec-privatekey.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
  00:91:d4:22:6f:4d:64:08:1f:a5:4a:d2:c5:49:3f:
  88:83:6a:d6:08:4c:fd:ec:27:fd:12:cd:ad:eb:91:
  e7:68:7f
pub:
  04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
  3a:b9:cb:fd:41:cf:43:4e:bf:09:75:c1:97:60:6c:
  47:77:3a:f4:26:dd:4d:22:76:98:13:41:86:0a:11:
  4c:4f:9a:4a:fb:07:ef:48:b9:18:97:9b:a5:d2:87:
  33:c1:62:a3:9e
ASN1 OID: secp256k1
```

类似地, 可采用下列命令查看公钥:

```
drequinox@drequinox-OP7010:~/Crypt$ openssl ec -in ec-pubkey.pem -pubin
-
text -noout
read EC key
Private-Key: (256 bit)
pub:
  04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
  3a:b9:cb:fd:41:cf:43:4e:bf:09:75:c1:97:60:6c:
  47:77:3a:f4:26:dd:4d:22:76:98:13:41:86:0a:11:
```



```

4c:4f:9a:4a:fb:07:ef:48:b9:18:97:9b:a5:d2:87:
33:c1:62:a3:9e
ASN1 OID: secp256k1
drequinox@drequinox-OP7010:~/Crypt$

```

除此之外，还可利用所需参数生成一个文件，在当前示例中为 SECP256K1，并进一步对其查看以理解底层参数的功能，如下所示：

```

~/Crypt$ openssl ecparam -name secp256k1 -out secp256k1.pem
drequinox@drequinox-OP7010:~/Crypt$ cat secp256k1.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----

```

该文件中包含了全部 SECP256K1 参数，并可通过下列命令加以分析：

```

drequinox@drequinox-OP7010:~/Crypt$ openssl ecparam -in secp256k1.pem
-text
-param_enc explicit -noout
Field Type: prime-field
Prime:
  00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
  ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
  ff:fc:2f
A: 0
B: 7 (0x7)
Generator (uncompressed):
  04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
  0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
  f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:
  0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:
  8f:fb:10:d4:b8
Order:
  00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
  ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
  36:41:41
Cofactor: 1 (0x1)

```

利用 SECP256K1 曲线域参数的生成器、阶数以及辅因子，上述示例显示了所用的素数以及 A、B 值。

除此之外，还存在另一种类型的加密原语，称为哈希函数。哈希函数不用于加密；相反，该函数生成一个固定长度的文本摘要。

3.4.3 密码原语

图 3.18 显示了密码原语的分类方式。

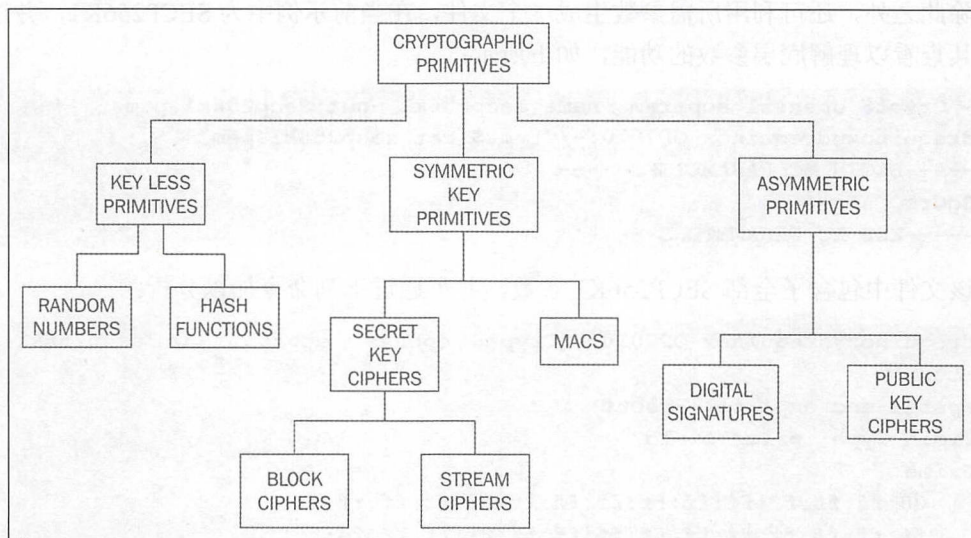


图 3.18 密码原语

3.4.4 哈希函数

哈希函数用于创建任意长度输入字符串的、固定长度的文摘。哈希函数并不需要密钥的参与，并可提供数据完整性服务，通常采用迭代和专用的哈希函数构建技术。对此，存在各种哈希函数族可供使用，如 MD、SHA1、SHA - 2、SHA - 3、RIPEMD 和 Whirlpool。哈希函数常用于数字签名和消息认证代码中，如 HMAC。一般具备 3 种安全性能，即抗原像性、抗第二原像性和抗冲突性。这些属性稍后在本节中解释。

哈希函数常用于提供数据完整性服务，并可用作单向函数，以构造其他密码原语，如 MAC 和数字签名。另外，一些应用程序使用哈希函数来生成伪随机数（PRNG）。哈希函数不需要使用到密钥。取决于完整性需求级别，哈希函数包含了多种不同的属性。

1. 将消息压缩为固定长度的文摘

该属性涉及以下操作：哈希函数必须能够接收任意长度的输入文本，并输出一个固定长度的压缩消息。哈希函数可生产各种位尺寸的压缩输出，通常在 128 位和 512 位之间。

2. 易于计算

哈希函数定义为高效且快速的单向函数——无论消息量大小如何，均可实现快速计算。如果消息量过大，计算效率可能会有所降低，但在实际应用过程中，该函数的计算速度仍可得到保障。

下面讨论哈希函数的安全性属性。

3. 抗原像性

考察下列方程：

$$h(x) = y$$

其中， h 表示哈希函数， x 表示输入， y 表示为哈希值。第一项安全属性要求 y 不能逆向计算至 x 。 x 则视为 y 的原像，因此命名抗原像性。这也称作单向属性。

4. 抗第二原像性

该属性要求：给定 x 和 $h(x)$ ，几乎无法获得任何其他消息 m 。其中， $m \neq x$ 且 m 的哈希值等于 x 的哈希值，即 $h(m) = h(x)$ 。这种性质也称为弱抗冲突性。

5. 抗冲突性

该属性要求：两个不同的输入消息不可散列至相同的输出结果。换句话说， $h(x) \neq h(z)$ 。这种特性也称为强抗冲突性。

考虑到哈希函数的性质，此类函数一般会存在某些冲突。相应地，两个不同的消息将散列至相同的输出结果中，但从计算角度上讲该过程一般是不可行的。哈希函数存在一个称为雪崩效应的概念：即使一个较小的变化，例如输入文本中一个字符的改变，也会导致完全不同的哈希输出。

哈希函数通常是由迭代哈希函数方法设计的。在这种方法中，输入消息以逐块、多轮方式压缩，最终生成压缩的输出结果。一种较为常见的迭代哈希函数类型是 Merkle-Damgard 结构。该结构将输入数据分割成相等大小的块，然后以迭代的方式通过压缩函数予以输入。压缩函数属性的抗冲突性保证了哈希输出也具有抗冲突特征。其中，压缩函数可以使用块密码来构建。除了 Merkle-Damgard 结构之外，一些研究人员也提出了其他一些压缩函数结构，例如，Miyaguchi-Preneel 和 Davies-Meyer。

后续内容还将讨论其他类型的哈希函数。

6. 信息文摘 (MD)

信息文摘功能在 20 世纪 90 年代早期非常流行，其中包括 MD4 和 MD5。需要说明的是，两者的 MD 功能均缺乏应有的安全性，且不推荐使用。MD5 是一个 128 位的哈希函数，通常用于文件完整性检查。

7. 安全哈希算法 (SHA)

SHA-0: NIST 于 1993 年发布的一个 160 位的函数。

SHA-1: NIST 随后发布了 SHA-1, 用于替代原有的 SHA-0。SHA-1 也是一种 160 位的哈希函数, 常用于 SSL 和 TLS 中。需要指出的是, SHA-1 现在也被认为是不安全的, 并被认证机构所弃用。同时, 在新的实现方案中, 也不建议使用 SHA-1。

SHA-2: 该类别包括由哈希位数定义的 4 个函数, 即 SHA-224、SHA-256、SHA-384 和 SHA-512。

SHA-3: 这也是最晚推出的 SHA 函数系列, 其中包括 SHA3-224、SHA3-256、SHA3-384 和 SHA3-512。SHA3 是 Keccak 的 NIST 标准版本, 采用了一种称为海绵构造的新方法, 而不是常用的 Merkle-Damgard 转换。

RIPEMD: RIPEMD 是“RACE 原始完整性校验消息摘要”的首字母缩略词, 基于 MD4 构建过程中的设计思想。RIPEMD 包含多个版本, 例如 128 位、160 位、256 位和 320 位。

Whirlpool: Whirlpool 是 Rijndael 密码的修正版本, 简称为 W, 并使用 Miyaguchi-Preneel 压缩函数。该函数是一种单向函数, 用于将两个固定长度的输入压缩成一个固定长度的输出, 同时也是一种单块长度压缩函数, 如图 3.19 所示。

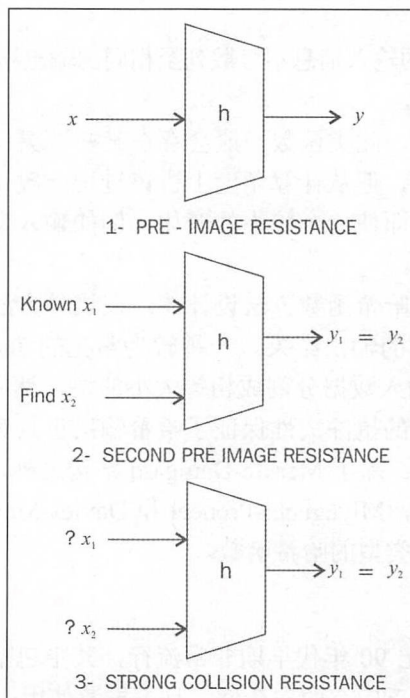


图 3.19 哈希函数的安全属性

哈希函数包含许多实际应用,包括加密协议和算法中简单的文件完整性检查和密码存储,常见于哈希表、分布式哈希表、Bloom 过滤器、指纹识别、P2P 文件共享等多种应用程序。

在区块链中,哈希函数扮演着非常重要的角色。特别是工作量证明函数两次使用了 SHA-256,以验证矿工所花费的计算工作量。RIPEMD 160 用于生成比特币地址。具体内容将在后续章节中详细讨论。

□ SHA 设计。后续内容将介绍 SHA-256 和 SHA-3 的设计过程。这两种方法分别用于比特币和 Ethereum 中。其中, Ethereum 并没有采用 NIST 标准 SHA-3,而是使用了 NIST 的原始算法。经过一些修正后,例如,增加了轮数和更简单的消息填充机制, NIST 将 Keccak 标准化为 SHA-3。

□ SHA256。SHA256 的输入消息大小小于 2^{64} 位。其中,块大小为 512 位,字大小 32 位,输出结果是 256 位的文摘。

压缩函数处理一个 512 位的消息块和一个 256 位的中间哈希值。该函数包含两个主要的组成部分:压缩函数和消息调度。

预处理算法按照如下方式工作:

- (1) 如果小于所要求的 512 位块尺寸,消息填充将采用 512 位的块长度。
- (2) 将消息解析为消息块,确保消息及其填充内容划分为 512 位的相等块。
- (3) 设置初始哈希值,表示为 8 个 32 位的字,这是通过前 8 个素数的平方根小数部分的前 32 位得到的。这一类初始值是随机选择的,以此初始化当前过程,并生成一个信用级别(算法中不包含后门)。

哈希计算步骤如下:

(1) 每个消息块依次进行处理,需要 64 轮来计算全部哈希输出结果。其中,每一轮均使用不同的常数,以确保没有两轮处于等同状态。

(2) 设置消息轮询。

(3) 初始化 8 个工作变量。

(4) 计算中间哈希值。

(5) 处理消息并生成输出哈希值,如图 3.20 所示。

图 3.20 中, a 、 b 、 c 、 d 、 e 、 f 、 g 、 h 分别表示寄存器; Maj 和 Ch 则采用位方式加以使用; \sum_0 和 \sum_1 按位执行旋转; W_j 和 K_j 则表示为轮常数并添加了 $\text{mod } 2^{32}$ 。

□ SHA3 设计(Keccak)。SHA-3 的结构与常见的 SHA-1 和 SHA-2 截然不同。SHA-3 背后的核心理念基于非密钥变换,这不同于其他常用的哈希函数结构(采用密钥变换)。另外,Keccak 也未采用 Merkle-Damgard 转换——在哈希函数中常用

于处理任意长度的输入消息。同时，Keccak 采用了一种较新的方案，称作海绵挤压结构，基本上可视为一种随机变换模型。除此之外，SHA-3 的不同版本也已标准化，例如 SHA3-224、SHA3-256、SHA3-384、SHA3-512、SHAKE128、SHAKE256。其中，SHAKE128 和 SHAKE256 表示为可扩展的输出函数，并由 NIST.XOF 予以标准化，以使输出结果可扩展为任意的期望长度。

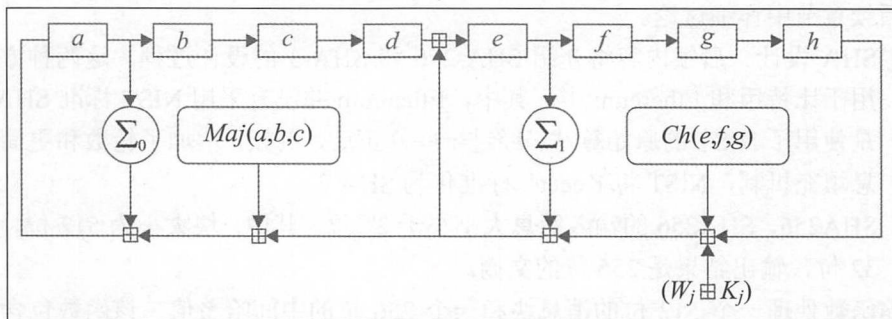


图 3.20 SHA 256 压缩函数中的一轮

图 3.21 显示了海绵和挤压模型，这也视为 SHA3 或 Keccak 的基础内容。就像海绵一样，首先，数据经填充后吸收到海绵中；通过 XOR 操作将其变成一个置换状态的子集，随后，源自海绵函数的输出结果经“挤压”后表示为转换后的状态。这里，比率表示为海绵函数的输入块大小，而容量则负责决定安全级别。

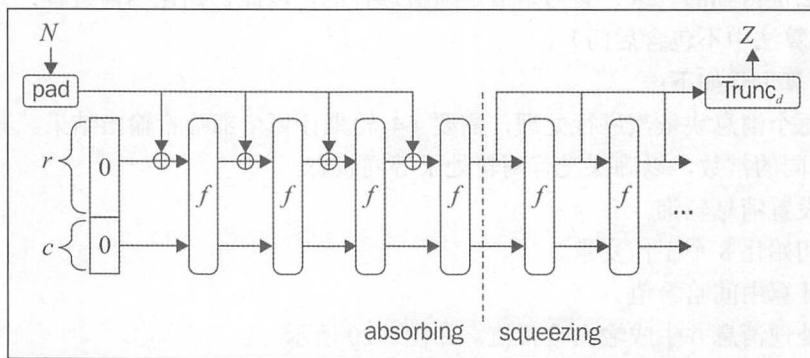


图 3.21 SHA-3 海绵和挤压函数

□ 哈希函数的 OpenSSL 示例。

当使用 SHA256 算法时，下面的命令生成一个 Hello 消息的 256 位哈希值：

```
~/Crypt$ echo -n 'Hello' | openssl dgst -sha256
(stdin)= 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
```


注意，即使是文本中出现很小的变化，例如更改 H，也会导致输出的哈希结果发生较大的变化。如前所述，这称为雪崩效应。具体操作如下所示：

```
:~/Crypt$ echo -n 'hello' | openssl dgst -sha256
(stdin)= 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

不难发现，两种输出结果截然不同，如下所示：

```
Hello:
18:5f:8d:b3:22:71:fe:25:f5:61:a6:fc:93:8b:2e:26:43:06:ec:30:4e:da:51:8
0:07:d1:76:48:26:38:19:69
hello:
2c:f2:4d:ba:5f:b0:a3:0e:26:e8:3b:2a:c5:b9:e2:9e:1b:16:1e:5c:1f:a7:42:5
e:73:04:33:62:93:8b:98:24
```

- ❑ 消息验证代码（MAC）。MAC 有时也被称作密钥哈希函数，并可提供消息完整性和身份验证功能。换句话说，MAC 负责提供数据源身份验证，这些都是使用发送方和接收方之间共享密钥的对称密码原语。另外，还可以使用块密码或哈希函数构造 MAC。
- ❑ 基于块密码的 MAC。在这种方法中，块密码用于密码块链接模式（CBC 模式），以生成一个 MAC。其中，可使用任意块密码，例如 CBC 模式中的 AES。实际上，消息的 MAC 视为 CBC 操作中最后一轮的输出结果。MAC 输出的长度与用来生成 MAC 的块密码的块长度是一样的。通过计算消息的 MAC，并将其与接收到的 MAC 进行比较，MAC 将得到验证，如果二者相同，则消息的完整性得到确认；否则，该消息视为已做更改。还应该注意，MAC 的工作方式类似于数字签名。然而，鉴于对称性质，它们均无法提供不可否认性服务。
- ❑ HMAC（基于哈希的 MAC）。与哈希函数类似，HMAC 产生一个固定长度的输出，并以任意长度的消息作为输入。在该方案中，发送者使用 MAC 对消息签名，接收者使用共享密钥验证该消息。通过秘密前缀或秘密后缀方法，消息密钥实现了哈希化。在第一种方法中，密钥与消息连接在一起，也就是说，密钥首先出现，随后是消息内容；而在后一种方法中，密钥位于消息之后，如下所示：

秘密前缀： $M = \text{MAC}_k(x) = h(k||x)$

秘密后缀： $M = \text{MAC}_k(x) = h(x||k)$

两种方法各有利弊，且已出现了对此的攻击行为。相应地，研究人员也提出了一些基于各类技术的 HMAC 构建方案，例如内部填充和外部填充，这一类方案在某些假设条件下可视为安全的，如图 3.22 所示。

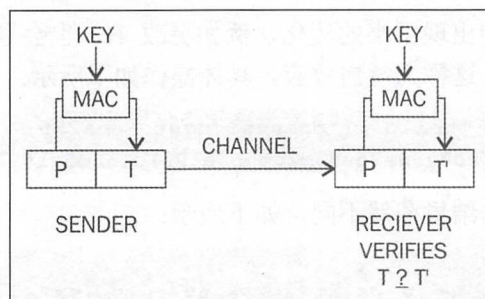


图 3.22 MAC 函数操作

8. Merkle 树

Merkle 树的概念由 Ralph Merkle 首先提出。图 3.23 展示了一种 Merkle 树的可视化结果，以方便读者理解这一概念。另外，Merkle 树允许对大数据集实现安全有效的验证。

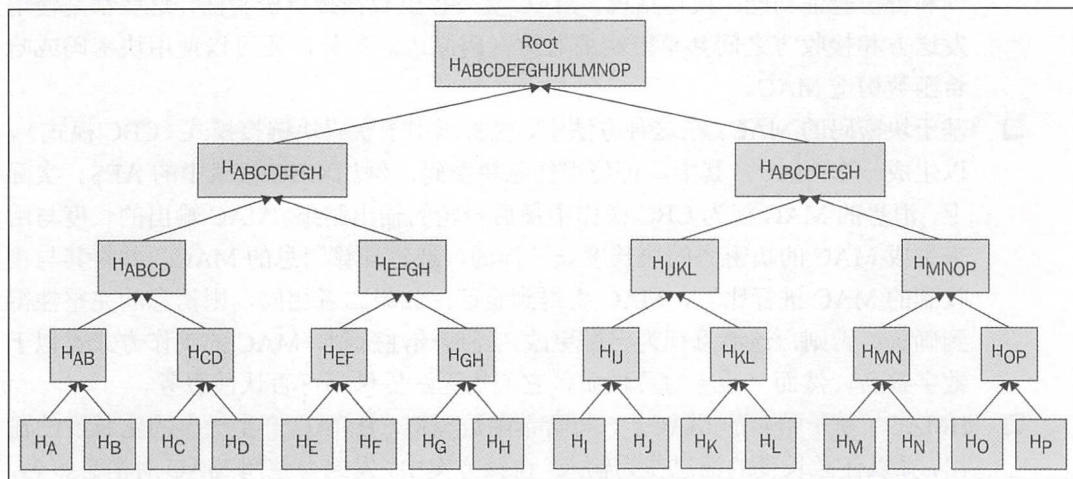


图 3.23 Merkle 树

Merkle 树是一棵二叉树，其中，输入信息置于叶节点中（不包含子节点的节点）；随后，子节点对中的数值集中实现哈希化，并生成父节点值，直至生成称作 Merkle 根节点的单一哈希值。

9. Patricia 树

为了理解 Patricia 树，首先引入一个称作前缀树的概念。前缀树或数位树这种数据结构定义为有序树，并可用于存储数据集。

Patricia 的全称是指用字母-数字编码来检索信息的实用算法。Patricia，也称为 Radix

树，是前缀树的一种紧凑表示。其中，如果某个节点为父节点的唯一子节点，则该节点将与父节点合并。

根据 Patricia 和 Merkle 的定义，Merkle-Patricia 树中涵盖了根节点，其中包含了整个数据结构的哈希值。

10. 分布式哈希表 (DHT)

哈希表是一种用于映射键-值的数据结构。在其内部，哈希函数用来计算桶数组的索引，从中可以获得所需值。桶中存储的记录使用了哈希键，并按特定的顺序加以组织。

根据上述定义，可以将分布式哈希表看作是一种数据结构。其中，数据分布于不同的节点上，而节点等价于一个 P2P 网络中的桶。

图 3.24 显示了 DHT 的工作原理。图中，数据通过一个哈希函数传递，从而生成一个紧凑的键。随后，该键将与 P2P 网络上的数据（值）联系起来。当网络上的用户请求数据（通过文件名）时，可再次对文件名进行哈希处理以生成同样的键；随后，在网络上的任何节点将被请求以找到相应的数据。除此之外，DHT 还提供了去中心化、容错和可扩展性等特征。

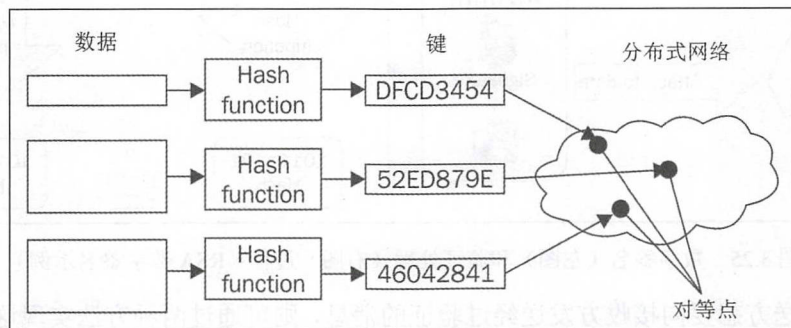


图 3.24 分布式哈希表

11. 数字签名

数字签名提供了一种将消息与消息源实体关联的方法。数字签名可用于实现数据源身份验证和不可否认性，并分两步进行计算。RSA 数字签名方案的操作步骤如下：

(1) 计算数据包的哈希值，这可视为数据完整性的一种保证，因为哈希值可以在接收者一端再次计算，并与原始哈希值匹配，以检测数据是否在传输过程中被修改。从技术上讲，消息签名可以在数据缺少哈希处理的情况下工作，但最终结果存在安全隐患。

(2) 利用签名者的私钥签订哈希值。由于只有签名者持有私钥，因而签名的真实性和签署的数据方可得到保证。

数字签名包含了一些较为重要的属性，例如真实性、不可伪造性和不可重用性。真实性意味着数字签名是由接收方验证的；不可伪造性确保仅消息的发送方能够通过私钥使用签名功能。换句话说，其他人无法生成合法发送者的签名消息；不可重用性意味着数字签名不能与消息分离并再次用于另一个消息。

图 3.25 显示了较为常见的数字签名功能。

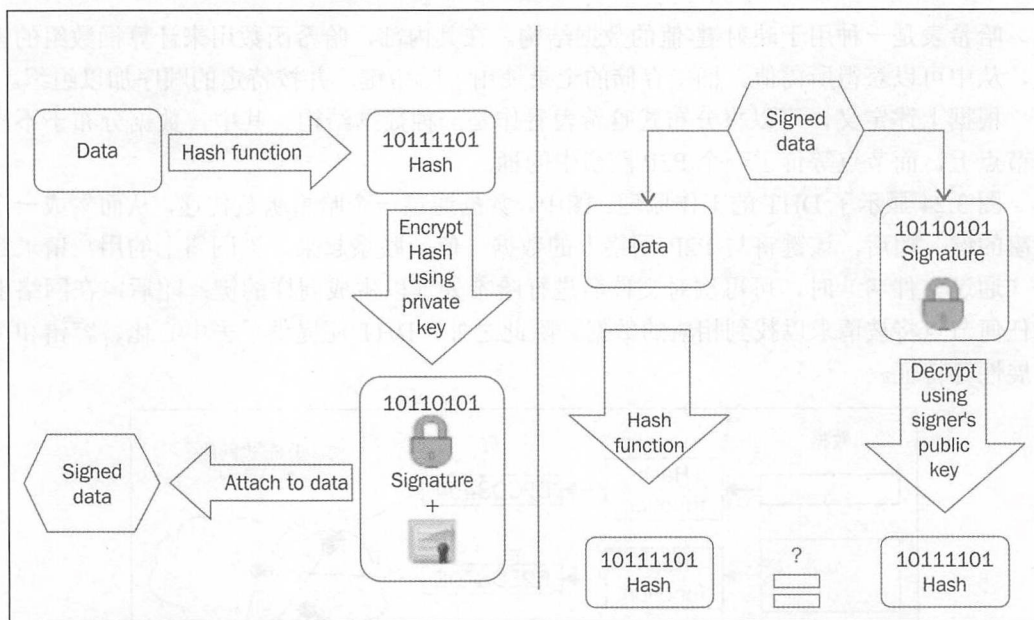


图 3.25 数字签名（左图）和验证处理（右图）过程（RSA 数字签名示例）

如果发送方想要向接收方发送经过验证的消息，则可通过两种方法实现这一项任务。下面介绍加密数字签名的两种应用方法。

- ❑ **签名-加密方法。**在这种方法中，发送者用私钥对数据进行数字签名，将签名附加到数据上，然后使用接收者的公钥对数据和数字签名进行加密。与接下来所描述的加密-签名方案相比，该方法是一种更安全的方案。
- ❑ **加密-签名方案。**在这种方法中，发送方使用接收方的公钥对数据进行加密，然后对加密数据进行数字签名。



注意：

在实际操作过程中，包含数字签名的数字证书由证书认证机构（CA）颁发，同时将公钥与身份相关联。

实际中可采用 RSA、数字签名算法、椭圆曲线数字签名算法等多种方案。RSA 是最常用的方法。然而，在椭圆曲线加密技术的推动作用下，基于 ECDSA 的方案也变得越来越流行。

下面将详细讨论 ECDSA 方案。

3.4.5 椭圆曲线数字签名算法 (ECDSA)

对于 ECDSA 方案的签名和验证操作，首选需要生成密钥对，相关步骤如下：

(1) 定义椭圆曲线 E ，其中包括模数 P ；系数 a 和 b ；生成器点 A ，并构成素数阶循环群 q 。

(2) 随机选取整数 d ，且有 $0 < d < q$ 。

(3) 计算公钥 B 以使 $B = dA$ 。

其中，公钥表示为如下 6 元组形式：

$$KPB = (P, A, B, q, A, B)$$

而步骤 (2) 中随机选取的 d 则表示为：

$$KPR = d$$

随后，可通过私钥和公钥生成签名，相关步骤如下：

(1) 选择一个临时的密钥 K_e ，其中 $0 < K_e < q$ 。同时确保 K_e 的随机性，且不存在两个签名具有相同的密钥；否则，可以计算私钥。

(2) 另一个值 R 是通过 $R = K_e A$ 来计算的，即 A (生成器点) 和临时随机密钥间的乘法运算。

(3) 利用点 R 的 x 坐标值初始化变量 r ，即 $r = xR$ 。

(4) 签名可按照下列方式计算：

$$S = (h(m) + dr)K_{e^{-1}} \bmod q$$

这里， m 表示需要计算签名的消息，而 $h(m)$ 则表示为消息 m 的哈希值。

签名验证则通过下列步骤完成：

(1) 辅助值 w 计算为 $w = s^{-1} \bmod q$ 。

(2) 辅助值 $u1 = w \cdot h(m) \bmod q$ 。

(3) 辅助值 $u2 = w \cdot r \bmod q$ 。

(4) 计算点 P ，且有 $P = u1A + u2B$ 。

(5) 执行验证过程。

(6) 如果步骤 (4) 中点 P 的 x 坐标等同于签名参数 $r \bmod q$ ，则 r, s 视为有效签名。也就是说：

$$XP = r \bmod q$$

表示为有效签名。

相应地

$$XP \neq r \bmod q$$

则表示为无效签名。

下面讨论如何使用 OpenSSL 生成、使用和验证 RSA 数字签名。

1. 生成数字签名

第一步是生成消息文件的哈希值，如下所示：

```
~/Crypt$ openssl dgst -sha256 message.txt
SHA256(message.txt)=
eb96d1f89812bf4967d9fb4ead128c3b787272b7be21dd2529278db1128d559c
```

其中，哈希生成和签名过程可在单一步骤中完成，如下所示。另外还需注意的是，privatekey.pem 是通过之前的某个步骤生成的。

```
~/Crypt$ openssl dgst -sha256 -sign privatekey.pem -out signature.bin
message.txt
```

下列内容显示了相关文件的字典。

```
~/Crypt$ ls -ltr
total 36
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 05:54 message.txt
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 05:57 message.bin
-rw-rw-r-- 1 drequinox drequinox 45 Sep 21 06:00 message.b64
-rw-rw-r-- 1 drequinox drequinox 32 Sep 21 06:16 message.ptx
-rw-rw-r-- 1 drequinox drequinox 916 Sep 21 06:28 privatekey.pem
-rw-rw-r-- 1 drequinox drequinox 272 Sep 21 06:30 publickey.pem
-rw-rw-r-- 1 drequinox drequinox 128 Sep 21 06:43 message.rsa
-rw-rw-r-- 1 drequinox drequinox 14 Sep 21 06:49 message.dec
-rw-rw-r-- 1 drequinox drequinox 128 Sep 21 07:05 signature.bin
~/Crypt$ cat signature.bin
v_[]h]h t + T~O1 s [( Cq"# A Q U, uf p* □ *7 T' u
eAy $ $ x <$ a ' : L qWh uG = $ $ : ~/Crypt$
```

为了对签名进行验证，可执行下列操作：

```
~/Crypt$ openssl dgst -sha256 -verify publickey.pem -signature
signature.bin message.txt
Verified OK
~/Crypt$
```


类似地，如果使用了某些无效文件，验证过程也随之失败，如下所示：

```
~/Crypt$ openssl dgst -sha256 -verify publickey.pem -signature
someothersignature.bin message.txt
Verification Failure
```

下面的例子将展示如何使用 OpenSSL 执行与 ECDSA 相关的操作。

2. 基于 OpenSSL 的 ECDSA

首先通过下列命令生成私钥：

```
~/Crypt$ openssl ecparam -genkey -name secp256k1 -noout -out
eccprivatekey.pem
~/Crypt$ cat eccprivatekey.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIMVmyrnEDOs7SYxS/AbXoIwqZqJ+gND9Z2/nQyzcpaPBoAcGBSuBBAAK
oUQDQgAEEKKS4E4+TATieBX8o2J6PxKkjcWwXPwNRo/k4Y/CZA4pXvlyTgH5LYm
QbU0qUtPM7dAEzOsaoXmetqB+6cM+Q==
-----END EC PRIVATE KEY-----
```

随后，从私钥中生成公钥，如下所示：

```
~/Crypt$ openssl ec -in eccprivatekey.pem -pubout -out eccpublickey.pem
read EC key
writing EC key
~/Crypt$ cat eccpublickey.pem
-----BEGIN PUBLIC KEY-----
MFYWEAYHKOZIZj0CAQYFK4EEAAoDQgAEEKKS4E4+TATieBX8o2J6PxKkjcWwXPw
NRo/k4Y/CZA4pXvlyTgH5LYmQbU0qUtPM7dAEzOsaoXmetqB+6cM+Q==
-----END PUBLIC KEY-----
~/Crypt$
```

假设名为 testsign.txt 的文件需要执行签名和验证操作，则可通过下列步骤完成：

(1) 生成测试文件：

```
~/Crypt$ echo testing > testsign.txt
~/Crypt$ cat testsign.txt
testing
```

(2) 运行下列命令，并使用 testsign.txt 文件的私钥生成签名：

```
~/Crypt$ openssl dgst -ecdsa-with-SHA1 -sign eccprivatekey.pem
testsign.txt > ecsign.bin
```

(3) 最后，运行下列验证命令：


```
~/Crypt$ openssl dgst -ecdsa-with-SHA1 -verify eccpublickey.pem  
-signature ecsign.bin testsign.txt  
Verified OK
```

另外，通过之前创建的私钥，还可生成证书，如下所示：

```
~/Crypt$ openssl req -new -key eccprivatekey.pem -x509 -nodes -days 365 -  
out ecccertificate.pem  
You are about to be asked to enter information that will be incorporated  
into your certificate request.  
What you are about to enter is what is called a Distinguished Name or a DN.  
There are quite a few fields but you can leave some blank  
For some fields there will be a default value,  
If you enter '.', the field will be left blank.  
-----  
Country Name (2 letter code) [AU]:GB  
State or Province Name (full name) [Some-State]:Cambridge  
Locality Name (eg, city) []:Cambridge  
Organization Name (eg, company) [Internet Widgits Pty  
Ltd]:Dr.Equinox!  
Organizational Unit Name (eg, section) []:NA  
Common Name (e.g. server FQDN or YOUR name) []:drequinox  
Email Address []:drequinox@drequinox.com
```

该证书可通过下列命令进行查看：

```
~/Crypt$ openssl x509 -in ecccertificate.pem -text -noout
```

对应结果如图 3.26 所示。

考虑到与区块链的相关性，以及未来区块链生态系统的潜在用途，密码学中还涵盖了其他主题内容。

3. 同态加密

通常，公钥密码系统（如 RSA）一般是乘法同态或加法同态的，如 Paillier 密码系统，该系统称为部分同态系统。加法 PHE 适用于电子投票和银行应用。当前，尚不存在相关系统可支持这两种操作。但在 2009 年，Craig Gentry 发现了一个完全同态的系统。由于这些方案允许对加密数据进行处理，且不需要解密，因此涵盖了多种不同的应用，特别是在需要维护隐私的情况下；同时，数据需要被潜在的、不受信任的各方进行处理，例如云计算和在线搜索引擎。近些年来，同态加密技术的发展前景十分广阔，在研究人员的努力下，其更高效性和实用性还将得到进一步的提升。这也是区块链技术中值得关注的方面，正如本书后面所描述的，此类技术可以解决区块链中保密和隐私问题。

```

drequinox@drequinox-OP7010: ~/Crypt
drequinox@drequinox-OP7010:~/Crypt$ openssl x509 -in ecccertificate.pem -text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 13205206053355364006 (0xb74250f0fc159ea6)
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: C=GB, ST=Cambridge, L=Cambridge, O=Dr.Equinox!, OU=NA, CN=drequinox/emailAddress=drequinox@drequinox.com
    Validity
      Not Before: Sep 27 00:09:43 2016 GMT
      Not After : Sep 27 00:09:43 2017 GMT
    Subject: C=GB, ST=Cambridge, L=Cambridge, O=Dr.Equinox!, OU=NA, CN=drequinox/emailAddress=drequinox@drequinox.com
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (256 bit)
      pub:
        04:10:a2:92:e0:4e:3e:4c:04:c8:78:15:fc:a3:62:
        7a:3f:12:a4:8d:ca:16:ad:73:f0:35:1a:3f:93:96:
        3f:09:90:38:a5:7b:e5:c9:38:07:e4:b6:26:41:b5:
        34:a9:4b:4f:33:b7:40:13:33:ac:6a:85:e6:7a:da:
        81:fb:a7:0c:f9
      ASN1 OID: secp256k1
    X509v3 extensions:
      X509v3 Subject Key Identifier:
        E1:68:E7:87:EE:E1:44:F0:70:71:76:4D:73:C6:15:14:F1:14:BE:F4
      X509v3 Authority Key Identifier:
        keyid:E1:68:E7:87:EE:E1:44:F0:70:71:76:4D:73:C6:15:14:F1:14:BE:F4

      X509v3 Basic Constraints:
        CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
    30:44:02:20:5e:ab:c9:85:f1:4f:e5:b1:05:e3:0f:ef:da:8d:
    d7:d5:5f:c5:e9:20:be:c3:3c:34:b6:74:f4:a6:5e:11:3c:e0:
    02:20:c6:5b:27:9a:78:c7:90:aa:cf:e9:42:c4:ac:da:fb:c8:7e:
    a0:15:62:0d:d0:89:e7:41:2a:03:9f:be:92:a7:2d:21
drequinox@drequinox-OP7010:~/Crypt$

```

图 3.26 X509 证书，使用基于 SHA-256 的 ECDSA 算法

4. 签密方案

签密方案是一种公共密钥加密原语，提供了数字签名和加密的所有功能，该方案是由 Yuliang Zheng 首先提出的，现在已成为 ISO 标准 ISO /IEC 29150:2011。从传统意义上讲，签名-加密或加密-签名方案具有不可伪造性、认证和不可否认性。但利用签密方案后，所有数字签名和加密服务囊括的成本低于签名-加密方案。

在单一逻辑步骤中，这可表示为成本（签名且加密）<<成本（签名）+成本（加密）。

5. 零知识证明

Goldwasser、Micali 和 Rackoff 首先提出了零知识证明理论，旨在证明一个断言的有效性，而不揭示任何关于断言的信息。ZKP 涉及 3 种属性，即完整性、可靠性和零知识属性。

完整性的意义在于，如果某个断言为真，则验证者确信证明者提供的断言。可靠性则可确保在断言为假时，不存在任何失信证明者可信验证者。顾名思义，“零知识产权”是零知识证明的关键属性，确保除了真或假之外，不会透露任何信息。

零知识证明在区块链中引起了研究人员的极大兴趣，其隐私属性在金融和其他许多领域，如法律和医学上都非常受欢迎。最近一个成功实施零知识证明机制的例子是 Zcash 加密货币。在 Zcash 中，实现了一种特定的零知识证明类型，即零知识简明非交互式知

识论证（ZK-Snark），这将在第 5 章中详细讨论。

6. 盲签名

盲签名是由 David Chaum 于 1982 年提出的，并基于公共密钥数字签名方案，如 RSA。盲签名背后的关键思想是，在没有实际显示消息的情况下，获取签名人所签署的消息。这是通过在签名之前伪装或使其不可见而实现的，因此得名“盲签名”。盲签名可以像普通数字签名一样对原始消息进行验证。作为一种机制，盲签名对数字现金方案的发展提供了支持。

7. 编码方案

除了加密原语之外，在各种场景中还会使用到二进制-文本编码方案。其中，最常见的应用是将二进制数据转换成文本，以便可以通过不支持二进制数据的协议进行处理、保存或传输。例如，有时，图像以 base64 编码的形式存储在数据库中，这也使得文本字段能够存储图片。一种常用的编码方案是 base64；另外，另一种名为 base58 的编码方案则是在比特币的推动下流行起来的。

密码学是一个巨大的领域，本节仅介绍了一些基本概念，这些概念对于理解密码学是必不可少的，特别是区块链和加密货币。下一节将介绍基本的金融市场概念。

下面考察与交易、交换和交易生命周期相关的常见术语，后续章节还将讨论与此相关的信息以及具体的用例。

3.5 金融市场和交易

金融市场的存在是为了方便储蓄者向投资者转移储蓄。经济体制中一般包含两个部门，即家庭和企业。金融市场的核心是充当储户与投资者之间的中介。相应地，基本上存在 3 种类型的市场，即货币市场、信贷市场和资本市场。货币市场是一个短期市场，其中，资金被借给公司或进行银行间的拆借。外汇或 FX 则是另一种货币市场。信贷市场主要由商业零售银行组成，并从中央银行借款，以抵押贷款或贷款形式向企业或家庭发放贷款。

资本市场促进金融工具的买卖，主要是指股票和债券。资本市场可分为两类：一级市场和二级市场。股票直接由公司向初级市场的投资者发行；而在二级市场，投资者通过证券交易所将其证券转售给投资者。其中，各交易所采用电子交易系统，以促进金融工具的交易。

3.5.1 交易

市场是交易者进行交易的地方，并可以提供相应的资产类别。

交易可以定义为：交易者买卖各种金融工具以产生利润和对冲风险的活动。投资者、借款人、对冲基金、资产交换者和赌徒都可视作不同类型的交易者。例如，当交易者处于亏欠状态时（如果交易员在购买合同时卖出了一份合同，并且在购买合同时持有多头仓位），则持有空仓。市场中存在多种交易方式。例如，经纪人或直接在交易所或柜台上交易。这里，经纪人是指为客户安排交易的代理人。经纪人代表客户以一定的价格或最好的价格进行交易。

3.5.2 交易所

交易所通常被认为是一个非常安全、规范、可靠的交易场所。最近，与传统的地板交易（floorbased trading）相比，电子交易获得了很高的人气。当前，交易员们将订单、价格和相关属性通过通信网络发布到所有相关的系统中，从而创建了一个虚拟市场。此外，外汇交易只能由交易员来进行，为了打破这些限制，交易方可以直接参与 OTC（场外）交易。

1. 订单和订单属性

订单一般指交易指令，是交易系统的主要组成部分，且具有以下一般属性：

- (1) 指令名。
- (2) 成交量。
- (3) 方向性（买入或卖出）。

(4) 订单的类型代表了不同的条件，例如限价订单和止损订单，其中一个例子是 1500 股苏格兰皇家银行普通股（15.50 英镑）。

订单以买入价和卖出价进行交易。通过将买入价和卖出价附于订单中，以体现交易者买入或卖出意向。相应地，交易者购买的价格称为买入价，交易者愿意卖出的价格称为卖出价。

2. 订单管理和传递系统

根据业务逻辑，订单传递系统向不同目的地分发订单，客户以此向经纪人发送订单。随后，可将这些订单发送给经销商、清算机构和交易所。

这里，存在两种不同类型的订单，最常见的是市场订单和限价订单。市场订单是指

以目前市面上最优惠的价格进行交易的指令，而这些订单会立即以现货价格进行交易。另一方面，限价订单是指以最好的价格进行交易的指令，但前提是不低于交易者设定的限价。这也可能取决于订单的方向，无论是卖出还是买入。所有这些订单均在订单簿中管理，这是一个由交易所维护的订单清单，记录了交易者买入或卖出的意向。

交易约定是指承诺出售或购买一定数量的金融工具，如证券、货币或大宗商品。交易者买卖的合约、证券、商品和货币通常称为交易工具，并在资产类别的大保护伞下运作。其中，最常见的类别是实物资产、金融资产、衍生品合约和保险合同。

3. 交易票证

交易票证表示与某项交易有关的所有细节的结合体，但会根据工具和资产类别的类型产生一些变化，稍后将对其各种属性加以讨论。



注意：

基础工具构成了交易的基础性内容，可以是货币、债券、利率、商品或股票。

4. 通用属性

通用属性包括一般的识别信息，以及与每一笔交易相关的基本特征，例如，包括唯一 ID、工具名称、类型、状态、交易日期和时间等。

5. 价值性

这一类属性均可视作与交易价值相关的特征，例如买卖价值、股票、交易、价格和数量。

6. 销售属性

销售属性指的是与销售相关的细节内容，例如销售人员的姓名。该属性仅表示为一个信息字段，通常不会对交易生命周期产生任何影响。

7. 交易对方

交易对方是交易过程中的重要组成部分，并显示了交易的另一方信息，并被要求成功地处理当前交易。其中，常用的属性包括对方的名称、地址、支付类型、引用 id、结算日期和交付类型。

3.5.3 交易的生命周期

通常，交易生命周期包括多种不同的阶段，例如下单、执行和结算。下面逐项加以

讨论:

- ☐ 预执行。在该阶段中执行下单操作。
- ☐ 执行和预定。当订单匹配并被执行时，将会转换成一项交易。在这个阶段，交易双方的合同到期。
- ☐ 确认行为，是指交易双方都同意的交易细节。
- ☐ 预定后的处理行为。这一阶段涉及各种审查和验证过程，以确定交易的正确性。
- ☐ 结算。这是交易中最重要的一部分，也是交易过程中的最后一个阶段。
- ☐ 当日处理。包括报告生成、损益计算和各种风险计算。
- ☐ 上述过程均会涉及相关人员和业务功能。通常情况下，这些功能被划分为交易、风险管理和清算等功能。

下一节将介绍一些概念，这些概念对于理解金融行业的各项规章制度是至关重要的。在讨论特定用例时，相关概念有助于读者理解所描述的场景。

3.5.4 订单预期者

这一类人员尝试在其他交易者之前赚取利润，通常熟知后者交易活动如何对价格产生影响，一般包括专家、以情感为导向的技术交易员。

3.5.5 市场操控

在英国和其他国家，市场操纵是完全非法的。欺诈交易者可以在市场上散布虚假信息，导致价格变动从而产生非法利润。通常，操纵市场行为是以贸易为基础的，包括广义的和时间特定的操作。期间，可能会造成股票的人为短缺、欺诈活动、价格操纵等。

上述内容均与金融犯罪有关，而且有可能形成区块链系统，进而扰乱市场。后续章节将对此以及相关示例予以详细讨论。

3.6 本章小结

本章旨在介绍密码学和金融市场中的相关概念，以便向读者提供相应的背景知识，从而理解后续章节中的相关内容。本章首先介绍了密码学的基本知识以及应用方案，如

对称和非对称加密。另外，还讨论了 OpenSSL 命令行的使用，读者可尝试各种命令，并亲身体验各种加密函数。此外，本章还提供了一些数学背景知识，特别是椭圆曲线密码学。本章所介绍的所有密码学概念都与区块链技术有关，涉及各种区块链、加密货币和相关生态系统。同时，本章还简要介绍了金融行业，并为分布式账本技术的各种示例设置了场景。由于密码学和金融学均涵盖了大量内容，因而本章仅对此予以简要介绍（少数内容例外）。在后续章节中，还会对特定的话题进行详细的扩展。

第4章 比特币

比特币是区块链技术的第一个应用，本章将详细介绍比特币技术。

比特币已经掀起了一场数字革命，并引入了第一个完全去中心化的数字货币，而且已经被证明是非常安全和稳定的。与此同时，这也激发了人们对学术和业界研究的浓厚兴趣，许多新的研究领域对此也有所借鉴。自2008年推出比特币以来，比特币人气颇高，目前已成为全球最成功的数字货币，并得到了数十亿美元的投资。比特币建立在密码学、数字现金和分布式计算领域的研究基础之上。在接下来的章节中，将简单介绍比特币的历史及其背景知识，以使读者了解比特币背后的相关内容。

几十年来，数字货币一直是较为活跃的研究领域。20世纪80年代早期，有人便提出了创建数字现金的建议。1982年，David Chaum制订了一项计划，使用盲签名来构建不可追踪的数字货币。在该方案中，银行通过用户签署的一个盲随机序列号来发行数字货币。随后，用户可将银行签署的数字令牌用作货币。该计划的限制性在于，银行必须跟踪所有使用的序列号。这可视为基于设计的中心系统，并受信于全部用户。1990年，David Chaum提出了一个名为e-cash的优化版本，不仅使用了盲签名，还提供了一些私人身份识别数据来传递随后发送给银行的信息。这一方案可检测出重复消费行为，但并没有对此进行阻止。也就是说，如果在两个不同的位置使用相同的令牌，则会显示重复消费行为人的身份标识。另外，e-cash只能体现固定数额的金钱。Adam Back于1997年推出的哈希现金，最初是用来阻止电子垃圾邮件的，其背后的理念是解决一类易于验证但难以计算的计算谜题。

具体而言，对于单一用户和某个电子邮件来说，额外的计算工作量并不明显；但是垃圾邮件发送者会感到沮丧，因为运行垃圾邮件所需的时间和资源将会大大增加。

b-money是Wei Dai于1998年提出的，引入了用工作量证明来创造货币的理念。该系统的一个主要弱点是，一个具备强大计算能力的对抗者可以主动提供未经请求的资金，同时，不允许网络调整至适当的难度级别。该系统缺乏关于节点之间共识机制的详细信息，而一些安全问题也没有得到妥善解决，如Sybil攻击。与此同时，Nick Szabo介绍了BitGold的概念，同样基于工作量证明机制。除了网络难度级别可调之外，BitGold与b-money面临相同的问题。Tomas Sander和Ammon TaShama在1999年推出了一个电子现金计划，并首次使用Merkle树代表货币，并通过零知识证明方案证实所持有的货币。该计划要求中央银行保留所有使用过的序列号记录。该方案允许用户完全匿名，但代价

是计算量的提升。RPOW（可重用的工作量证明）是由 Hal Finney 在 2004 年提出的，并使用 Adam Back 的哈希现金作为用于创建资金的、计算资源的证明。另外，这可视为一类中央系统，保存了一个中央数据库，并跟踪所有使用过的 POW 令牌。同时，这也是一个使用远程认证的在线系统，通过一个可信的计算平台（TPM 硬件）实现。

4.1 比特币概述

2008 年，中本聪（Satoshi Nakamoto）发表了一篇关于比特币的论文，即“Bitcoin: A Peer-to-Peer Electronic Cash System”。论文中引入的第一个关键思想即 P2P 电子现金，同时需要一个中介银行在对等成员间转移支付。

比特币源自多年来的加密领域中的研究成果，如 Merkle 树、哈希函数、公钥加密和数字签名等。此外，BitGold、b-money、哈希先进和加密时间戳等概念为比特币的发明奠定了基础。所有这些技术巧妙地结合在一起，创造了世界上第一种去中心化的货币。比特币中所解决的关键问题是，如何优雅地处理拜占庭将军问题，以及双重支出问题。

自 2012 年以来，比特币的价值显著增加，如图 4.1 所示。

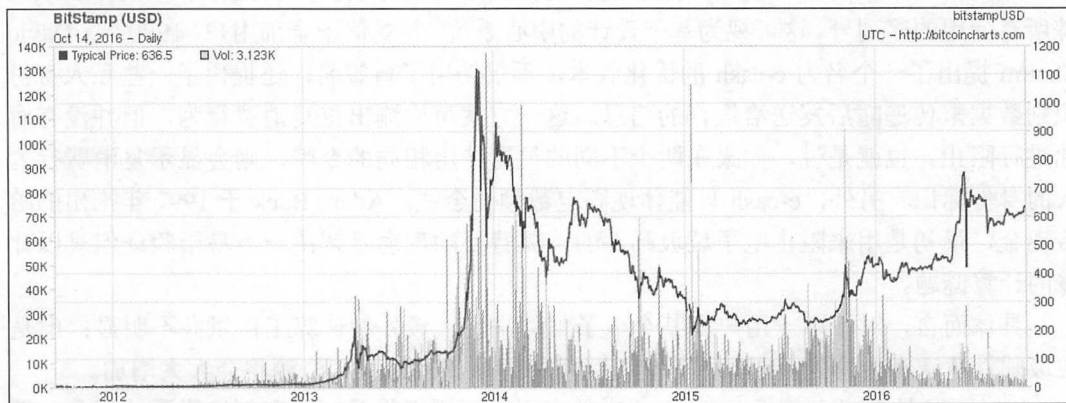


图 4.1 自 2012 年以来比特币的价值（对数规模）

对比特币的监管一直是一个有争议的话题，就像自由主义者的梦想一样，执法机构和政府都在制订各种监管措施，例如，纽约州金融服务管理局发布的 BitLicense，即是颁发给那些执行与虚拟货币相关活动的企业许可证。

比特币的增长也得益于所谓的网络效应，也称需求方经济规模。从概念上讲，基本上意味着随着网络用户的不断增加，其价值也就越高。随着时间的推移，比特币网络的增长呈现为指数增长。尽管比特币的价格波动很大，但在过去几年里，比特币的价格总

体上仍处于大幅上涨趋势。目前（在本书编写时），比特币的价格是 815 英镑。

4.1.1 比特币的概念

比特币可以通过多种方式来定义：它是一个协议，一种数字货币，一个平台。另外，作为 P2P 网络、协议和软件的组合，还促进了数字货币比特币的创建和使用。请注意，使用大写 B 的比特币是用来指代比特币协议的，而使用小写 b 的比特币则用来指代比特币。相应地，P2P 网络中的节点使用比特币协议彼此通信。

比特币的发明首次实现了货币的去中心化。此外，比特币的双重支出问题以一种优雅而巧妙的方式得到了解决。例如，当用户在同一时间向两个不同的用户发送货币时，即会产生双重支出问题，并且作为有效事物进行独立验证。

4.1.2 密钥和地址

椭圆曲线密码技术用于在比特币网络中生成公钥和私钥对。比特币地址的创建方式可描述为：使用一个私钥的相应公钥，并对其执行两次哈希操作。其中，首先可使用 SHA256 算法，然后使用 RIPEMD160 算法。最终的 160 位哈希值通过一个版本号添加前缀，最后采用 Base58Check 编码方案进行编码。比特币地址表示为 26~35 个字符，以数字 1 或 3 开始。一个典型的比特币地址看起来像一个字符串，如下所示：

1ANAgUGG8bikEv2fYsTBnRUmx7QUcK58wt

除此之外，另一种常见的编码方式则是二维码，上述地址的二维码形式如图 4.2 所示。



图 4.2 比特币地址 1ANAgUGG8bikEv2fYsTBnRUmx7QUcK58wt 的二维码形式

目前，比特币存在两种类型的地址，即常用的 P2PKH 和 P2SH 类型，且分别以 1 和 3 开始。早期，比特币使用了公钥直接支付方式，现在则被 P2PKH 取代。然而，对于币基（coinbase）地址，比特币中仍采用公钥直接支付这种方式。另外，地址不应多次使用，否则将产生隐私和安全问题。无效地址复用在一定程度上规避了匿名问题，然而比特币仍存在一些其他方面的安全问题，如事务的延展性，这需要通过不同的方案加以解决。

图 4.3 显示了源自 bitaddress.org 的私钥和比特币地址。

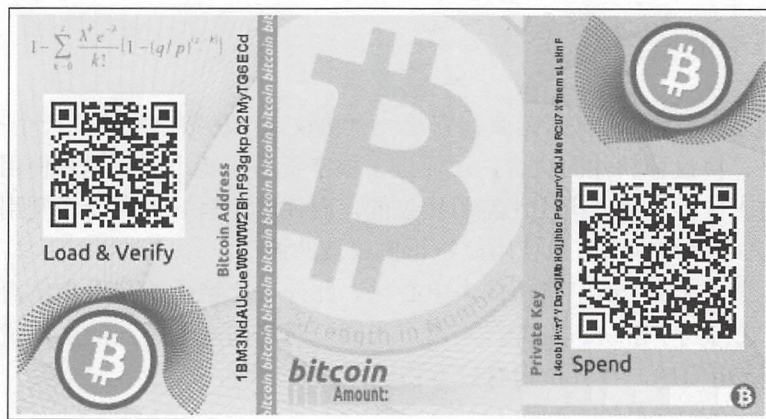


图 4.3 源自 bitaddress.org 的私钥和比特币地址

4.1.3 比特币中的公钥

在公钥加密过程中，公钥由私钥生成。这里，比特币使用基于 SECP256K1 标准的 ECC。其中，私钥被随机选择，长度为 256 位。相应地，公钥可以通过未压缩或压缩格式显示。基本上讲，公钥表示为椭圆曲线上的 x 和 y 坐标，在未压缩格式中，包含了十六进制格式中的 04 前缀。 X 和 Y 坐标都是 32 位的长度。总的来说，与非压缩格式的 65 个字节相比，压缩公钥的长度为 33 字节。公钥的压缩版本基本上只包含 X 部分，因为 Y 部分可以据此派生出来。公钥压缩版本工作原理可描述为，比特币客户端最初使用的是未压缩密钥，但从比特币核心客户端 0.6 开始，压缩密钥被用作标准。

密钥被各种前缀所识别，如下所示：

- ❑ 非压缩公钥采用 0x04 作为前缀。
- ❑ 如果公钥 32 位的 y 部分为奇数，则压缩公钥始于 0x03。
- ❑ 如果公钥 32 位的 y 部分为偶数，则压缩公钥始于 0x02。

具体来讲，如果可对 ECC 图形予以可视化显示，则 y 坐标可分别在 x 轴的上方或下方。考虑到曲线的对称性，只有在素数场中的相关位置才需要被存储。

4.1.4 比特币中的私钥

在 SECP256K1 ECDSA 所推荐的范围内，私钥一般表示为 256 位数字。作为有效的私钥，随机选取的 256 位数字位于 0x1~0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF

BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140 范围内。

私钥通常使用钱包导入格式 (WIF) 进行编码, 以使其易于复制和使用。同时, WIF 可以转换为私钥, 反之亦然。

此外, 有时还使用迷你私钥格式, 并在 30 个字符以内对密钥进行编码, 以实现空间有限的存储操作, 例如, 蚀刻的硬币或损坏的二维码。除此之外, 比特币核心客户端还允许加密包含私钥的钱包。

4.1.5 比特币货币单位

图 4.4 显示了比特币货币单位。其中, 最小的比特币面额为 Satoshi。

DENOMINATION	ABBREVIATION	FAMILIAR NAME	VALUE IN BTC
Satoshi	SAT	Satoshi	0.00000001 BTC
Microbit	μBTC (uBTC)	Microbitcoin or Bit	0.000001 BTC
Millibit	mBTC	Millibitcoin	0.001 BTC
Centibit	cBTC	Centibitcoin	0.01 BTC
Decibit	dBTC	Decibitcoin	0.1 BTC
Bitcoin	BTC	Bitcoin	1 BTC
DecaBit	daBTC	Decabitcoin	10 BTC
Hectobit	hBTC	Hectobitcoin	100 BTC
Kilobit	kBTC	Kilobitcoin	1000 BTC
Megabit	MBTC	Megabitcoin	1000000 BTC

图 4.4 比特币货币单位

4.1.6 Base58Check 编码

这种编码方式可防止不同字符之间出现混淆, 例如, 在不同字体中, 0011 可能看起来相同。编码基本上采用二进制字节数组, 并转换为人类可读的字符串。该字符串由一组 58 个字母数字符号组成。关于更多的解释和逻辑, 读者可参考比特币源代码中的

base58.h 源文件, 如图 4.5 所示。

```
6  /**
7   * Why base-58 instead of standard base-64 encoding?
8   * - Don't want 00Il characters that look the same in some fonts and
9   *   could be used to create visually identical looking data.
10  * - A string with non-alphanumeric characters is not as easily accepted as input.
11  * - E-mail usually won't line-break if there's no punctuation to break at.
12  * - Double-clicking selects the whole string as one word if it's all alphanumeric.
13  */
14  #ifndef BITCOIN_BASE58_H
```

图 4.5 比特币源代码的解释

另外, 比特币地址则采用 Base58check 进行编码。

4.1.7 虚地址

鉴于比特币地址基于 base58 编码, 因此可以生成包含人类可读信息的地址, 如图 4.6 所示。

虚地址则是用纯蛮力法生成的, 相关示例如图 4.7 所示。



图 4.6 采用二维码编码的公共地址



图 4.7 生成自 <https://bitcoinvanitygen.com/> 的虚地址

4.2 交易/事务

交易/事务是比特币生态系统的核心。取决于具体的需求任务，交易可以简单到向某一比特币地址发送比特币，或者也可以变得十分复杂。其中，每项交易至少由一项输入和输出组成。这里，输入可视为之前交易所创建的消费货币。如果某项交易正在“铸造”新货币，则不存在输入内容，因此也就不需要签名。如果某项交易正在向其他用户（一个比特币地址）发送货币，则需要发送方利用私钥进行签名；同时，为了显示这些货币的来源，还需要对之前的交易进行引用。事实上，货币是 Satoshi 中所表示的、未动用的交易输出结果。

在区块链中，交易并未加密且公开可见。另外，区块是由交易组成的，并可以通过在线区块链浏览器进行查看。

4.2.1 交易的生命周期

交易的生命周期涉及以下内容：

- （1）用户/发送者使用钱包软件或其他接口发送事务。
- （2）钱包软件使用发送者的私钥来签署事务。
- （3）该交易通过扩散算法向比特币网络进行广播。
- （4）挖掘节点将在下一个挖掘块中包含该事务。
- （5）一旦矿工解决了工作量证明问题，并将最新挖掘的新区块传播至网络中，即可开始采矿过程。本章稍后将对工作量证明加以解释。
- （6）节点对区块进行验证并进一步传播该区块，同时开始生成确认。
- （7）最后，确认结果出现于接收者的钱包中，在大约 6 次确认之后，交易被认为已得到确认。这里，数字 6 只是一个推荐方案。即使在第一次确认之后，交易也可视为终止。此处重点在于，在 6 次确认之后，实际上消除了重复支出的概率。

4.2.2 交易的结构

在较高的级别上，交易包含元数据、输入和输出，经适当整合后将创建一个区块。交易结构如表 4.1 所示。

表 4.1 交易结构

字 段	尺 寸	描 述
版本号	4 字节	对于交易处理过程，用于制定矿工和节点的规则
输入计数器	1~9 个字节	交易中所包含的输入量
输入列表	变量	每个输入都由多个字段组成，包括之前的交易哈希值、之前的 Txout 索引、Txin-脚本长度、Txin-脚本和可选的序列号。区块中的第一项交易也称为币基（coinbase）交易，并指定一个或多个交易输入
输出计数器	1~9 个字节	表示输出量的正整数值
输出列表	变量	交易中的输出
lock_time	4 字节	定义了有效交易的最早时间，可以是 UNIX 中的时间戳，或者区块号

- ❑ 元数据：交易中的这一部分内容包含一些相关值，例如交易的大小、输入和输出的数量、交易的哈希值和 lock_time 字段。另外，每项交易均包含一个指定版本号的前缀。
- ❑ 输入：通常，每项输入都会消费上一次的输出结果。同时，每项输出均视为未动用的交易输出（UTXO），直至输入消耗该输出。
- ❑ 输出：输出仅涉及两个字段，包含了比特币的发送指令。其中，第一个字段表示 Satoshis 的数量，而第二个字段则是一个锁定脚本，该脚本涵盖了需要满足的各项条件，以便使用输出结果。关于交易的锁定、解锁以及输出，后续内容将对此进行讨论。
- ❑ 验证：验证过程通过交易的脚本语言执行。

图 4.8 显示了一个交易示例。

1. 脚本语言

比特币使用一种简单的、基于堆栈的语言来描述比特币应用和传输方式。该语言缺少图灵完备性且不包含训话结构，以避免在比特币网络上长期运行/悬挂脚本时产生的不良影响。这种脚本语言基于一种类似于 Forth 的语法，并使用一个逆波兰标记。其中，每个操作数均由其操作符执行，同时按照从左到右的顺序，并使用一种“后进先出”（LIFO）堆栈进行评估。

脚本使用各种操作码或指令来定义操作。操作码也称作字、命令或函数。早期版本的比特币节点中包含了操作码。然而，鉴于其设计问题，这一类操作码已不再使用。

脚本代码的各种类型一般定义为常量、流控制、堆栈、位逻辑、连接和算术、加密和锁定时间。

```
{
  "txid": "08af7960ca9255c67686296fb65452ed3f96f18831c9a3d8ea552e4ccee5c4af",
  "hash": "08af7960ca9255c67686296fb65452ed3f96f18831c9a3d8ea552e4ccee5c4af",
  "size": 226,
  "vsize": 226,
  "version": 1,
  "locktime": 969523,
  "vin": [
    {
      "txid": "3e553260a0a94860f7043eb6576e15e6cf2990aea961210ae1fde328bb08b0",
      "vout": 1,
      "scriptSig": {
        "asm":
"3045022100cfb31edabc62c82b41d12f651d2e3e013ee1a7ee2bb4526f3dda640e6d8d224502207d8d1d8e41350b9cdf36f389f942ab68c12f113fe99014f5d6df6610407877d2[ALL] 037bc82d0078993f6943e7ff6e82e82da600f34edc8bca136331a9901c8bb60b0d",
        "hex":
"483045022100cfb31edabc62c82b41d12f651d2e3e013ee1a7ee2bb4526f3dda640e6d8d224502207d8d1d8e41350b9cdf36f389f942ab68c12f113fe99014f5d6df6610407877d20121037bc82d0078993f6943e7ff6e82e82da600f34edc8bca136331a9901c8bb60b0d"
      },
      "sequence": 4294967294
    }
  ],
  "vout": [
    {
      "value": 2.30000000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 07e78644a61343068fa8d4940a79976e758ac6ef OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a91407e78644a61343068fa8d4940a79976e758ac6ef88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "mgEkNzxV3qbYtDKEKTvo1VpgJ63Au619q2"
        ]
      }
    }
  ]
}
```

图 4.8 解码后的交易示例，其中包含了各种字段

事务脚本可通过整合 ScriptSig 和 ScriptPubKey 进行评估。其中，ScriptSig 表示为解锁脚本，而 ScriptPubKey 则表示为锁定脚本。这也体现了交易事务的评估方式。首先，事务将被解锁并被消费。相应地，ScriptSig 则通过希望解锁当前事务的用户提供。ScriptPubkey 体现了交易输出结果中的部分内容，同时制定了须满足的相关条件以及消费输出结果。换言之，输出被包含相关条件的 ScriptPubKey（锁定脚本）锁定；当满足条件时即解锁输出内容，并于随后兑取货币。

2. 常用的操作码

全部操作码均声明在比特币客户端源代码中的 script.h 文件中。读者可访问 <https://github.com/bitcoin/bitcoin/blob/master/src/script/script.h>，并查看下列注释下方的内容：

```
/** Script opcodes */
```

表 4.2 中列出了常见的操作码描述。该表取自比特币开发者指南。

表 4.2 常见的操作码描述

操 作 码	描 述
OP_CHECKSIG	采用公钥和签名，并验证交易哈希值的签名。若匹配，则将 TRUE 压入至栈中；否则将 FALSE 压入至栈中
OP_EQUAL	若输入间相等则返回 1；否则返回 0
OP_DUP	复制栈中最顶端的数据项
OP_HASH160	输入内容执行两次哈希操作：第一次采用 SHA-256，第二次采用 RIPEMD-160
OP_VERIFY	若最顶端栈值为假，则将交易事务标记为无效
OP_EQUALVERIFY	等同于 OP_EQUAL，但随后运行 OP_VERIFY
OP_CHECKMULTISIG	获取第一个签名并将其与各公钥比较，直至匹配。重复该过程直到全部签名均被检测。若所有签名均为有效，则返回值 1 作为结果；否则返回 0

4.2.3 交易类型

比特币中存在多种可用的脚本，可以处理从源到目的地之间的值传输。根据交易事务的需求条件，此类脚本的复杂度也不一而同。下面讨论标准交易事务类型，该类型事务使用 `IsStandard()` 和 `IsStandardTx()` 测试进行评估——只有通过测试的标准事务允许在比特币网络中进行挖掘或广播。尽管如此，非标准事务在网络上依然是有效和允许的。

- ❑ 公钥哈希值支付（P2PKH）：P2PKH 是最为常用的交易事务类型，并用于向比特币地址发送事务。交易事务的对应格式如下：

```
ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY
OP_CHECKSIG

ScriptSig: <sig> <pubKey>
```

其中，`ScriptPubKey` 和 `ScriptSig` 连接后一并执行。稍后还将展示一个示例，并对此进行详细解释。

- ❑ 脚本哈希值支付（P2SH）：P2SH 用于将交易事务发送至脚本哈希值（也就是说始于 3 的地址），并在 BIP16 中被标准化。除了传递脚本之外，还将对 `redeemScript`（兑取脚本）进行评估并确保有效，对应模板如下：

```
ScriptPubKey: OP_HASH160 <redeemScriptHash> OP_EQUAL

ScriptSig: [<sig>...<sign>] <redeemScript>
```

- ❑ **MultiSig (MultiSig 支付)**: M(n)多签名事务脚本是一种复杂类型的脚本。其中, 所构造的脚本需要多个签名才能生效, 并以此兑取交易事务。通过这一类脚本, 可构建各种复杂的事务, 如托管和存款。对应模板如下:

```
ScriptPubKey: <m> <pubKey> [<pubKey> . . . ] <n> OP_CHECKMULTISIG
ScriptSig: 0 [<sig> . . . <sign>]
```

原始的 Multisig 已不再被使用, Multisig 一般是 P2SH 兑取脚本中的部分内容, 之前曾对此有所提及。

- ❑ **Pubkey 支付**: 该脚本较为简单, 常用于 Coinbase 交易事务中。目前, 这一类型的脚本已经过时, 且仅用于早期版本的比特币中。在当前示例中, 公钥存储在脚本中, 而解锁脚本需要使用私钥签署事务。对应模板如下:

```
<PubKey> OP_CHECKSIG
```

- ❑ **Null 数据/OP_RETURN**: 该脚本用于存储区块链中的任意数据, 并收取一定的费用。对应消息限制为 40 个字节。由于 OP_RETURN 在任何情况下均不会通过验证, 因而脚本的输出结果不可兑取。在当前示例中, ScriptSig 未被要求。对应的模板也较为简单, 如下所示:

```
OP_RETURN <data>
```

图 4.9 显示了 P2PKH 脚本执行过程。

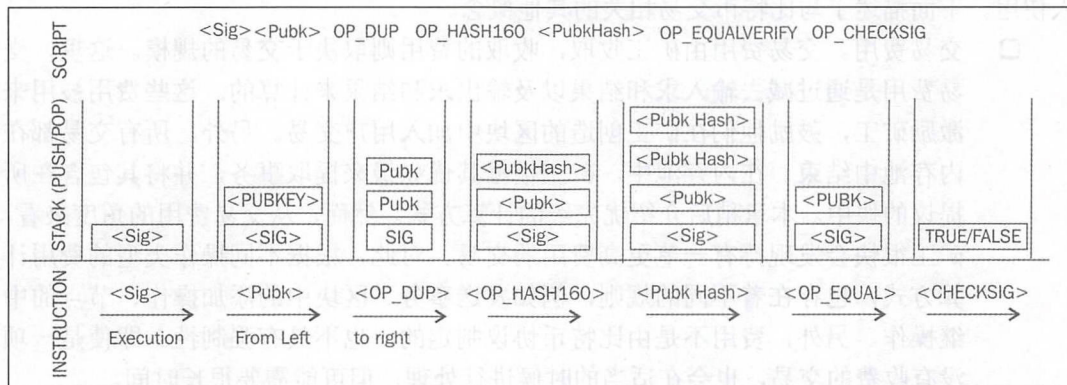


图 4.9 P2PKH 脚本执行过程

所有的交易事务最终都被编码为十六进制, 然后通过比特币网络传输。当使用比特币测试网络上的比特币客户端时, 下列命令可检索示例事务:


```
drequinox@drequinox-OP7010:~$ bitcoin-cli --testnet getrawtransaction  
"08af7960ca9255c67686296fb65452ed3f96f18831c9a3d8ea552e4ccee5c4af"  
0100000001b008bb28e3fde10a2161a9ae9029ebcfe6156e57b63e04f76048a9a06032  
553e010000006b483045022100cfb31edabc62c82b41d12f651d2e3e013ee1a7ee2bb4  
526f3dda640e6d8d224502207d8d1d8e41350b9cdf36f389f942ab68c12f113fe99014  
f5d6df6610407877d20121037bc82d0078993f6943e7ff6e82e82da600f34edc8bca13  
6331a9901c8bb60b0dfefefffff028085b50d000000001976a91407e78644a61343068f  
a8d4940a79976e758ac6ef88ac95bddd1c000000001976a914dad770cccb1026ebf87a  
cacfe35f2d6f2d336faa88ac33cb0e00
```

1. Coinbase 交易

Coinbase 一般通过矿工生成，往往是区块中的第一个交易事务，同时可用于生成新货币。Coinbase 包含了一个特定的字段，即 `coinbase`，作为 Coinbase 交易的输入。该交易最多支持 100 个字节，并可用于存储任意数据。关于创始区块，《泰晤士报》曾刊登了下列著名的评论：

“财政大臣第二次对银行实施紧急救市——

——泰晤士报，2009 年 1 月 3 日

该消息也从另一个角度证实了以下事实：创始块的挖掘时间不会早于 2009 年 1 月 3 日。

2. 什么是 UTXO

UTXO (Unspent Transaction Output) 是指未动用的交易输出，可以作为新交易的输入使用。下面描述了与比特币交易相关的其他概念。

- ❑ 交易费用。交易费用由矿工收取，收取的费用则取决于交易的规模。这里，交易费用是通过减去输入求和结果以及输出求和结果来计算的。这些费用被用来激励矿工，鼓励他们在矿工创造的区块中加入用户交易。另外，所有交易都在内存池中结束。在内存池中，矿工根据其优先级来提取事务，并将其包含在所提议的块中。本章稍后介绍优先级的计算方式。然而，从交易费用的角度来看，矿工很快会发现持有一笔更高费用的交易。对此，根据不同操作类型的费用计算方式，也存在着不同的规则，例如发送事务、区块中的添加操作、节点的中继操作。另外，费用不是由比特币协议制定的，也不具有强制性。即使是一项没有收费的交易，也会在适当的时候进行处理，但可能需要很长时间。
- ❑ 合约。正如在比特币核心开发者指南中所定义的，合约基本上是使用比特币系统来执行金融协议的交易。虽然这是一个相对简单的定义，但却具有深远的影响——允许用户设计复杂的契约，同时可以在许多现实场景中使用。通过合约，可开发一个完全去中心化的、独立的、减少风险的平台。针对于此，可以使用

比特币脚本语言构建各种合约，如托管、仲裁和微支付通道。当前，脚本实现尚存在某些局限性，但是仍然可尝试开发各种类型的合约。例如，对于多方签署交易后的资金发放，或者一段时期后的资金发放，才会发行基金。这两种场景可以采用 multiSig 和交易锁定时间选项实现。

- ❑ 交易的延展性。由于比特币实现过程中的 bug，比特币的交易延展性也被同时引入。针对这一问题，交易 ID 可能会被篡改，进而导致某项交易貌似未被执行，从而产生重复存款或取款等问题。换句话说，该 bug 可在确认比特币交易之前改变其唯一 ID。

也就是说，如果在确认之前更改了 ID，那么交易似乎并没有发生，随后即可发起双重存款或取款攻击。

- ❑ 交易池。交易池也称为内存池，基本上是通过节点于本地内存中创建的，以维持一个临时的交易列表，这些交易尚未在区块中得到确认。在经过验证和优先级操作后，交易被包含在一个区块中。
- ❑ 交易验证。验证过程由比特币节点执行，下列内容取自比特币开发者指南中的描述：

- 对语法进行检测，确保交易语法的正确性。
- 对输入和输出进行验证，确保二者非空。
- 检查字节尺寸是否小于最大区块尺寸（当前为 1MB）。
- 输出值应位于所允许的货币范围内（当前为 0~2100 万 BTC）。
- 除了 Coinbase 交易之外，所有输入均需包含上一次输出内容，且不应被转发。
- 对 nLockTime 进行验证，且不可超过 31 位。对于有效交易，nLockTime 不应小于 100 字节。另外，在标准签名中，签名操作数的数量应小于或不超过 2。
- 拒绝非标准事务。例如，ScriptSig 只允许在堆栈上推送数字。ScriptPubkey 不会通过 isStandard() 检查。
- 如果交易池或主分支某个区块中已存在一项匹配的交易，则应拒绝该项交易。
- 如果每项输入的引用输出在池中的其他交易中存在，则该交易将被拒绝。
- 对于每项输入，必须存在一个引用的输出交易。在主分支和事务池中对其进行搜索，以查找任何输入是否缺少输出交易，这可被视为一个孤儿（orphan）交易。如果一项匹配的交易不在池中，将被添加到孤儿交易池中。
- 对于每项输入，如果引用的输出交易表示为 Coinbase，必须至少包含 100 次确认；否则，交易将被拒绝。
- 对于每项输入，如果引用的输出不存在或已被消费，则该交易将被拒绝。
- 当使用被引用的输出交易来获取输入值时，应验证每个输入值以及总和是

否位于 0~2100 万 BTC 这一允许范围内。

- 如果输入值的总和小于输出值的总和，则拒绝该交易。
- 如果交易费用太低，以至于无法添加至空块中，则拒绝该交易。

4.3 区 块 链

区块链是在比特币网络上的、所有交易的时间戳、有序和不可变列表的公共账本。其中，每个区块通过链中的哈希值标识，通过引用前一区块的哈希值链接至前一个区块中。

下面首先介绍区块的结构，其中描述了区块头；随后还将考察区块链结构。

4.3.1 区块链结构

表 4.3 显示了区块链的结构。

表 4.3 区块链结构

字 节	名 称	描 述
80	区块头	包含了源自区块头的多个字段
变量	交易计数器	该字段涵盖了区块中的全部交易数量，包括 Coinbase 交易
变量	交易	区块中的全部交易

4.3.2 区块头结构

表 4.4 显示了区块头结构。

表 4.4 区块头结构

字 节	名 称	描 述
4	版本	区块的版本号，表示须遵守的区块验证规则
32	前一区块头的哈希值	前一区块头的双重 SHA256 哈希值
32	Merkle 根哈希值	区块中全部交易中 Merkle 树的双重 SHA256 哈希值
4	时间戳	该字段包含了 UNIX 时间格式的、区块的创建时间。更为准确地讲，该字段表示为矿工开始对区块头执行哈希操作的时间
4	难度目标	表示区块的难度目标
4	Nonce	矿工可重复调整的任意值，并生成一个哈希值，以满足难度目标阈值

区块链表示为一种块链结构,如图 4.10 所示。其中,通过引用前一个区块头的哈希值,每一个区块可链接至前一个区块中。这种链接方式可确保交易无法被修改,除非记录该交易的区块,以及随后的全部区块均被修改。另外,第一个区块不存在前区块,因而称作创始区块。

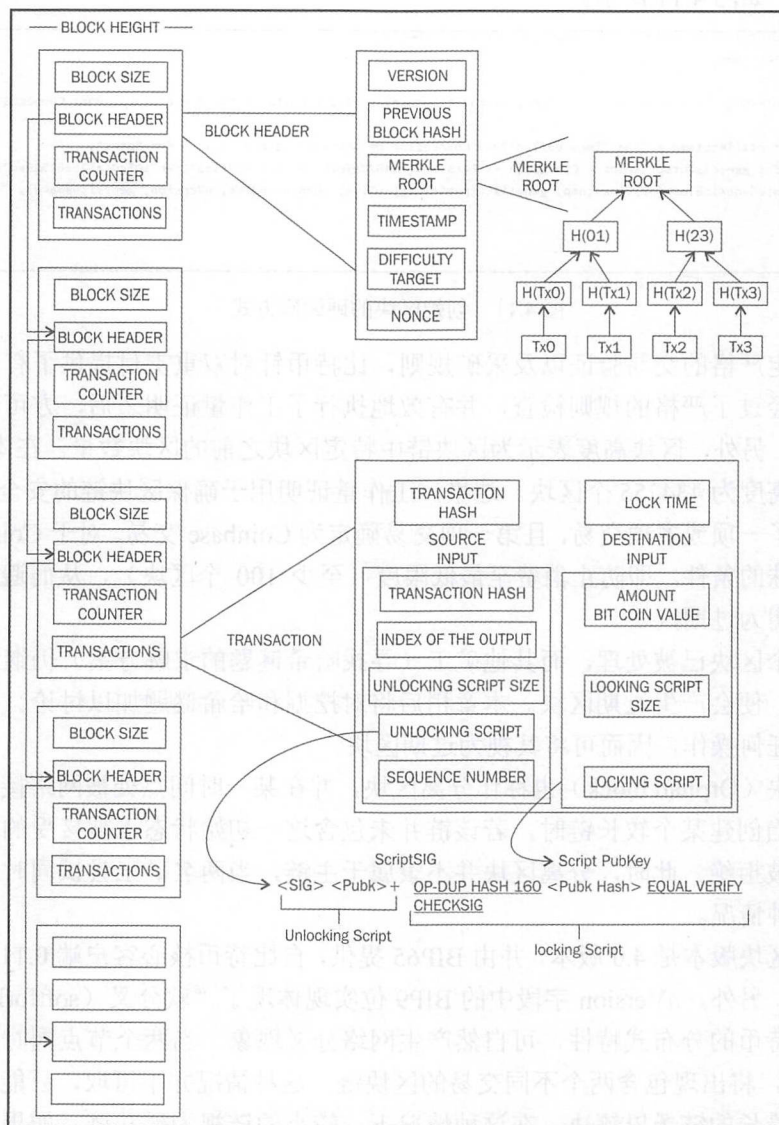


图 4.10 区块链、区块、区块头、交易和脚本

4.3.3 创始区块

创始区块表示比特币区块链的第一个区块。在比特币核心软件中，创始区块采用了硬编码方式，如图 4.11 所示。

```

48 * CtxOut(nValue=50.00000000, scriptPubKey=0x5F1DF16828764C8A578D08)
49 * vMerkleTree: 4a5e1e
50 */
51 static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce, uint32_t nBits, int32_t nVersion, const CAmount& genesisReward)
52 {
53     const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks";
54     const CScript genesisOutputScript = CScript() << ParseHex("04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ealf61deb649");
55     return CreateGenesisBlock(pszTimestamp, genesisOutputScript, nTime, nNonce, nBits, nVersion, genesisReward);
56 }
57
58 /**

```

图 4.11 创始区块的硬编码方式

通过制定严格的交易验证以及采矿规则，比特币针对双重支付提供了有效的保障机制。只有在经过了严格的规则检查，并有效地执行了工作量证明之后，方可将情况添加至区块链中。另外，区块高度表示为区块链中特定区块之前的区块数量。在本书编写时，当前区块链高度为 434755 个区块。这里，工作量证明用于确保区块链的安全性。其中，各区块包含了一项或多项交易，且第一项交易确定为 Coinbase 交易。对于 Coinbase 交易，存在一项特殊的条件，即防止消费至最低限度（至少 100 个区块），从而避免在后续过程中将其声明为过期区块。

当某一个区块已被处理，而其他矿工（寻找哈希谜题的求解方案）仍继续尝试对该区块操作时，便会产生过期区块。本章稍后将对挖掘和哈希谜题加以讨论。鉴于不再对该区块实施任何操作，因而可将其视为过期区块。

孤儿区块（Orphan block）也称作分离区块，并在某一时间点处被网络接受为有效区块；但是，当创建某个较长链时，若该链并未包含这一初始状态下被接受的区块，那么这一区块将被拒绝。此时，分离区块并不隶属于主链，当两名矿工尝试同时生成区块时即会出现这种情况。

最新的区块版本是 4.0 版本，并由 BIP65 提供，自比特币核心客户端 0.11.2 发布以来即投入使用。另外，nVersion 字段中的 BIP9 位实现体现了“软分叉（softfork）”变化。

由于比特币的分布式特性，可自然产生网络分叉现象。当两个节点同时获取一个有效的 b 锁时，将出现包含两个不同交易的区块链。这种情况并不可取，只能通过比特币网络并接受最长的链予以解决。在这种情况下，较小的链视为孤儿链。如果对方设法获得了网络 hashrate（计算能力）51% 的控制，即可获得自身的交易历史版本。

区块链的分叉可以随着比特币协议的改变而出现。在软分叉的情况下，只有之前的有效区块不再被接受，从而使软分叉向后兼容。对于软分叉，只有矿工需要升级到新的客户机软件时才能使用新的协议规则；而规划中的升级行为并不一定会创建分叉，因为所有用户应该已经更新完毕了。另一方面，硬分叉将使之前有效区块处于失效状态，并要求所有用户升级。某些时候，新的交易类型将作为软分叉而被添加；另外，任何更改均会导致硬分叉，例如块结构的更改或主协议的更改。

截至 2017 年 2 月 4 日，比特币区块链的当前规模约为 101GB。图 4.12 显示了区块链基于时间函数的尺度增加状态。

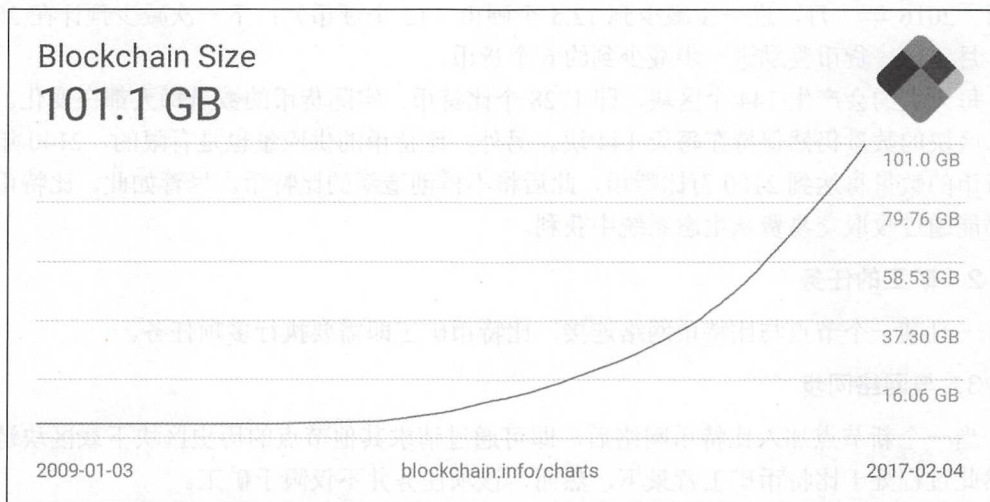


图 4.12 截至 2017 年 2 月 4 日，区块链的当前尺度规模

每隔 10 分钟，新的区块链就会被添加到原区块链中；而网络难度则每隔 256 个区块即会动态调整一次，以保持网络中新区块的稳定增长。

相应地，网络难度通过下列方程计算：

$$\text{目标} = \text{前一目标} \times \text{时间} / 2016 \times 10 \text{ 分钟}$$

难度和目标之间是可以互换的，二者代表着相同的事物。其中，先前的目标表示旧的目标值；此处，时间表示生成 2016 个区块所消耗的时间。另外，网络难度基本上意味着矿工发现一个新区块的难易程度，即哈希谜题的难度。

下面讨论挖掘（采矿）过程，进而解释如何求解哈希谜题。

1. 挖掘过程

挖掘行为是一种资源密集型过程，并以此将新区块添加到区块链中。通过节点挖掘

过程，区块中包含了验证后的交易，并添加到区块链中。这一过程是资源密集型的，确保矿工消耗了所需的资源，以使区块被接受。同时，新货币是矿工们花费了所需的计算资源后生成的。这也保护了系统免受欺诈和双重支出攻击，同时为比特币生态系统增加更多的虚拟货币。

大约每 10 分钟就会产生一个新的区块。矿工们在创造新区块时，将会得到货币奖励，交易费用则以纳入区块中的交易作为回报。相应地，新的区块以接近固定速率的速度创建。此外，新比特币的创造速度每隔 210000 个区块就会减少 50%，大约每 4 年一次。当比特币最初被引入时，相应的奖励是 50 个比特币；在 2012 年，这一奖励减少到 25 个比特币；2016 年 7 月，进一步减少到 12.5 个比特币（12 个比特币）；下一次减少预计在 2020 年 7 月 4 日，货币奖励进一步减少到约 6 个比特币。

每天大约会产生 144 个区块，即 1728 个比特币。实际货币的数量每天都在变化。然而，区块的数量仍然保持在每天 144 块。另外，比特币的供应量也是有限的，2140 年，比特币的数量将达到 2100 万比特币，此后将不再创造新的比特币。尽管如此，比特币矿工仍能通过收取交易费从生态系统中获利。

2. 矿工的任务

一旦某一个节点与比特币网络连接，比特币矿工即需要执行多项任务。

3. 与网络同步

当一个新节点加入比特币网络后，即可通过请求其他节点的历史区块下载区块链。虽然此过程处于比特币矿工背景下，然而，该项任务并不仅限于矿工。

- ❑ 交易验证：通过完整节点的验证、签名验证以及输出结果，在网络上传播的交易将被验证。
- ❑ 区块验证：通过对某些规则进行评估，矿工和完整的节点即可开始验证所收到的区块，包括对区块中每项交易的验证，以及对 nonce 值的验证。
- ❑ 创建新区块：待对网络上传播的消息进行验证后，通过对其进行整合即可生成一个新区块。
- ❑ 执行工作量证明：这项任务也视为挖掘过程的核心内容，矿工通过解决计算谜题获取一个有效区块。其中，区块头包含一个 32 位的 nonce 字段，在结果哈希值小于预定目标之前，需要多次更改 nonce。
- ❑ 获取奖励：一旦某个节点解决了哈希谜题，则立即广播结果，其他节点对此予以验证并接受该区块。某些时候，鉴于可能与同时发现的另一个区块产生冲突，新造区块将不会被其他矿工接受。然而，一旦区块被接受，矿工将获得 12.5 比

代币（2016 年）和相关交易费用。

4. 工作量证明

工作量证明（PoW）是指已经花费了足够的计算资源来构建一个有效的区块。PoW 基于如下理念：每次都选择一个随机节点来创建一个新的区块。在该模型中，节点相互竞争并按与计算能力呈正比的方式进行选择。下式总结了比特币工作量证明过程：

$$H(N \parallel P_hash \parallel Tx \parallel Tx \parallel \dots Tx) < Target$$

其中，N 表示 nonce，P_hash 是前一区块的哈希值，Tx 表示区块中的交易事务，Target 则表示目标网络的难度值。这意味着，之前提到的连接字段的哈希值应该小于目标哈希值。另外，获取 nonce 的唯一方法是采用蛮力法。一旦矿工遇到具有特定模式的、一定数量的零值，对应区块就会立即被传播并被其他矿工所接受。

5. 挖掘算法

挖掘算法包含以下步骤：

- ❑ 从比特币网络中获取之前的哈希区块。
 - ❑ 将一组传播于网络上的潜在交易“组装”为一个区块。
 - ❑ 通过 SHA256 算法，利用 nonce 和前一个哈希值计算区块头的双哈希值。
 - ❑ 如果结果哈希值小于当前难度级别（目标），则终止当前处理过程。
 - ❑ 如果结果哈希值大于当前难度级别（目标），则通过递增 nonce 来重复此过程。
- 随着比特币网络哈希率的增加，32 位的 nonce 总数量很快就消耗殆尽。为了解决这一问题，可考察附加 nonce 这一解决方案，将 Coinbase 交易作为额外的 nonce 来源，以提供更大范围的 nonce，以便被矿工搜索。
- ❑ 鉴于一段时间内挖掘难度的增加，采用单 CPU 笔记本电脑挖掘的比特币，现在需要通过专用的挖掘中心解决哈希谜题。当前难度级别可采用比特币命令行界面进行查询，如下所示：

```
$ bitcoin-cli getdifficulty
258522748404.5154
```

getdifficulty 命令的返回值如图 4.13 所示。

6. 哈希速率

哈希速率基本上表示每秒哈希值的计算速率。在比特币的早期，计算速率通常较小，这一点与 CPU 十分相似；目前则配备了专用的矿池和 ASIC，并在过去的几年中呈指数级增长。同时，这也导致了计算难度的增加。图 4.14 显示的哈希速率图展示了随时间增加的哈希速率，并在 Exa 哈希值中度量。这意味着，在 1 秒钟内，比特币网络的矿工们

将计算超过 10000000000000000000 次的哈希值。

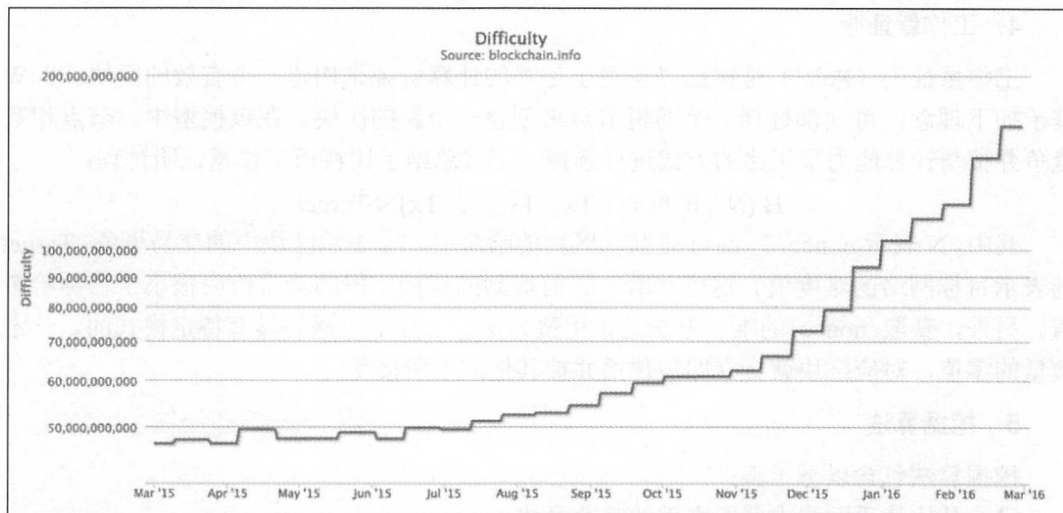


图 4.13 一段时间内的挖掘难度

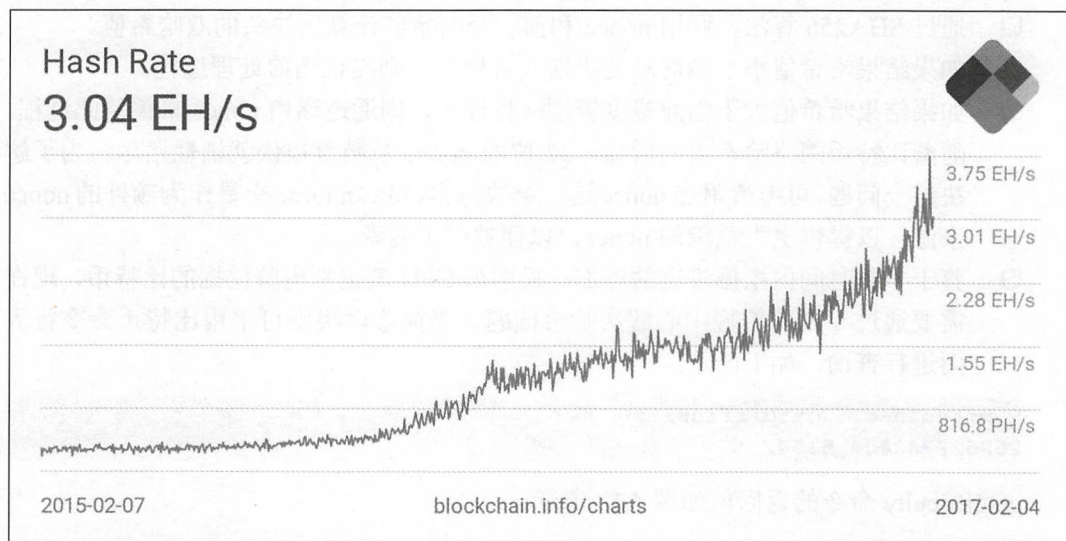


图 4.14 两年中的哈希速率（截至 2017 年 6 月 2 日）

7. 挖掘系统

随着时间的推移，比特币矿工使用了各种方法来开采比特币。由于挖掘的核心内容是基于双 SHA256 算法，矿工已经开发出复杂的系统以更快地计算哈希值。下列内容回

顾了比特币中使用的不同类型的挖掘方法，以及它们是如何随时间演变的。

8. CPU

在最初的比特币客户端中，CPU 挖掘是第一种可用的挖掘类型，用户甚至可以使用笔记本电脑或台式电脑来挖掘比特币。目前，CPU 采掘不再有利可图，而是转向更先进的采矿方法，例如基于 ASIC 的采矿。

9. GPU

鉴于比特币网络的难度增加，寻找更快的方法便成了一种普遍趋势，矿工们开始使用 PC 上的 GPU 或图形卡进行挖掘。GPU 支持更快的并行计算，通常使用 OpenCL 语言编程。与 CPU 相比，这是一个更快的选择方案。此外，其他一些技术，如超频，还可进一步提升 GPU 的计算能力。对于比特币挖掘，多显卡应用进一步扩展了其应用范围。然而，GPU 采矿也存在一些限制，如过热问题，以及需要专门的主板和额外的硬件以安装多个显卡。

10. FPGA

GPU 采矿并未持续很长时间，矿工们很快找到了基于 FPGA 进行采矿的另一种方法。现场可编程门阵列（Field Programmable Gate Array, FPGA）基本上是一个集成电路，可以编程来执行特定的操作。FPGA 通常在硬件描述语言（HDL）中编程，如 Verilog 和 VHDL。双 SHA256 很快成为 FPGA 程序员眼中一项极具吸引力的编程算法，并发布了一些开源项目。与 GPU 相比，FPGA 提供了更好的性能。然而，诸如可访问性、编程难度、程序设计的专业知识以及配置 FPGA 等问题导致了 FPGA 生命短暂。此外，ASIC 的出现也加快了 FPGA 挖掘系统的淘汰速度。在 FPGA 开采有利可图时，市场上也出现了一批挖掘硬件，X6500 矿工、Ztex 和 Icarus 等。一些 FPGA 制造商，如 Xilinx 和 Altera，也推出了一批 FPGA 硬件和开发板，并可用于对挖掘算法进行编程。

11. ASIC

专用集成电路（Application Specific Integrated Circuit, ASIC）的设计旨在执行 SHA256 操作。各家制造商均推出了相应的专业芯片，并提供了很高的哈希速率。这在一段时期内反映良好，但随着挖掘难度级别的不断提升，单击版的 ASIC 逐渐势微。

目前，挖掘行为已经超出了个人范畴。当前，使用数千个 ASIC 并行单元的专业挖掘中心正在向用户提供采矿合约。单用户根本无法并行运行数千个 ASIC——这需要启用专用的数据中心和硬件，其成本令人望而却步。

图 4.15 显示了上述 4 种挖掘类型。

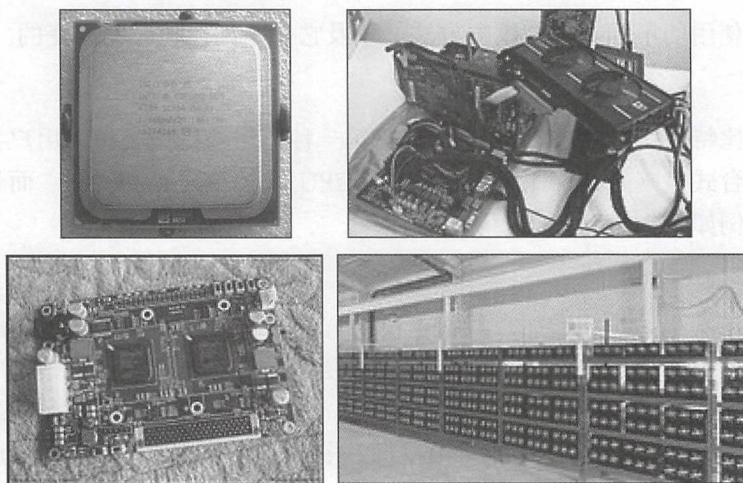


图 4.15 4 种挖掘类型

12. 矿池

当一组矿工一起挖掘一个区块时，即形成了矿池。如果区块被成功挖掘，池管理器接收 Coinbase 交易，然后负责将奖励分配给投资于该区块的矿工组。与独立挖掘相比，该机制更趋于盈利。对于前者，仅存在唯一的矿工试图求解部分哈希逆函数（即哈希谜题）。在矿池中，奖励是支付给每个成员的，而不管该成员（具体而言，个人节点）是否解决了当前问题。

此处，矿池管理者可使用多种模式用来支付给矿工，例如，份额支付模型以及比例模型。在份额支付模型中，矿池管理者向参与挖掘的全部矿工支付固定费用；而在比例模型中，该份额是根据求解哈希谜题时所消耗的计算资源量制定的。

当前存在大量的商业矿池，通过云以及易于使用的 Web 界面提供挖掘服务合同。其中，最常用的矿池包括 AntPool、F2Pool 和 BW.COM。针对某些主要的挖掘工具，图 4.16 显示了哈希计算能力间的比较结果。

当生成超出比特币网络 51% 的哈希速率，并试图控制超过 51% 的网络时，则挖掘中心系统将会出现问题。如前所述，51% 的攻击可产生双重支付工具，并对共识机制产生影响。实际上，这将在比特币网络上强加另一个版本的交易历史。

上述问题在比特币历史上曾发生过一次，当时 GHash.IO（一个大型的矿池）设法获得了超过 51% 的网络容量。对此，学术界已经提出了理论解决方案，例如，两阶段的工作量证明，以抑制大型矿池。这个方案引入了第二个加密谜题，结果在挖掘池中显示了它们的私钥，或者提供了挖掘池相当一部分哈希速率，从而减少了矿池的整体哈希速率。

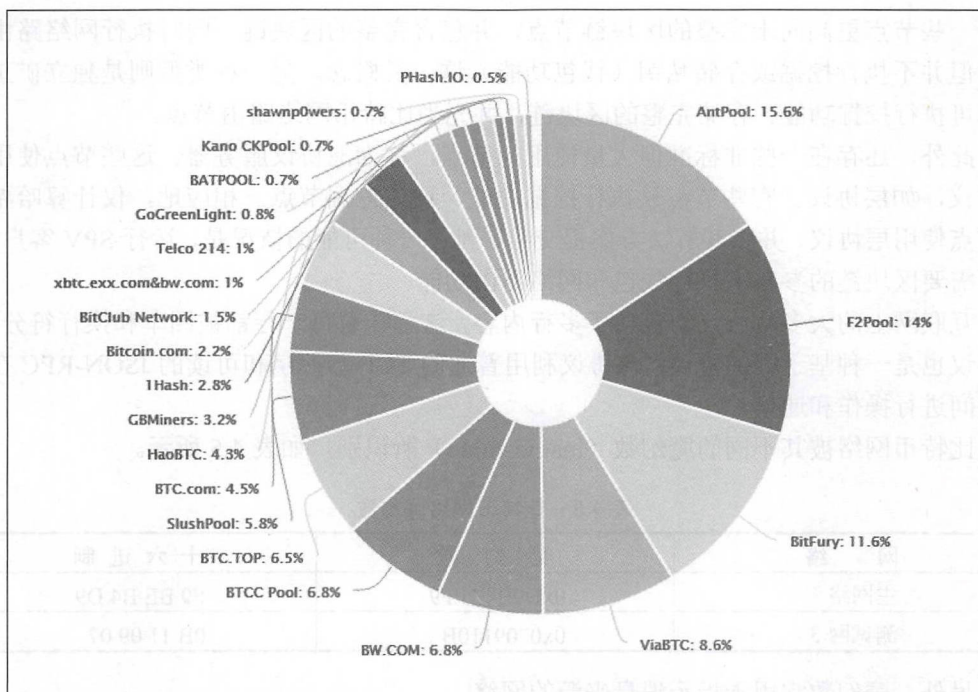


图 4.16 挖掘工具及其哈希计算能力（熟悉速率）。

该图源自 <https://blockchain.info/pools>，时间截至 2017 年 6 月 2 日

当前，市场上存在各种类型的硬件可用于采矿。目前，最赚钱的是 ASIC；大量的供应商也在源源不断地提供专业的硬件设施。除非花费大量的金钱和精力来构建个人采矿平台甚至是采矿中心，否则独立采矿难以获取利润。鉴于目前的困难因素（截至 2016 年 10 月），如果某个用户可提供 12TH/s 的哈希速率，则每天可赚取 0.01366887BTC（大约 8 美元）。与设备投资相比，这一数字是非常低的。当纳入其他成本后，例如电力等运营成本，最终往往无法获取收益。

4.3.4 比特币网络

比特币网络是一个 P2P 网络，其节点负责交换交易和区块。另外，网络上存在不同类型的节点，其中涉及两种主要类型的节点，即完整节点和 SPV 节点。顾名思义，完整节点是执行比特币、矿工、全区块链存储和网络路由功能的比特币核心客户端的实现结果。实际上，并无必要执行所有这些功能。SPV 节点或轻量级客户端只执行钱包和网络路由功能。最新版本的比特币协议是 70014，并随比特币核心客户端 0.13.0 被引入。

一些节点更倾向于完整的区块链节点，并包含完整的区块链，同时执行网络路由功能，但并不执行挖掘或存储私钥（钱包功能）这一类概念。另一种类型则是独立矿工节点，可执行挖掘功能、存储完整的区块链，并充当比特币网络路由节点。

此外，还存在一些非标准但大量使用的节点，称为池协议服务器。这些节点使用备选协议，如层协议。有些节点只执行挖掘功能，称为挖掘节点。相应地，仅计算哈希值的节点使用层协议，并将其解决方案提交给矿池。一种可能的情况是，运行 SPV 客户端，在不需要区块链的参与下执行钱包和网络路由功能。

互联网上的大多数协议均包含了多行内容，这意味着每一行都被回车和换行符分隔。层协议也是一种基于行的协议，该协议利用普通的 TCP 套接字和可读的 JSON-RPC 在节点之间进行操作和通信。

比特币网络被其不同的魔幻数（magic value）所识别，如表 4.5 所示。

表 4.5 比特币网络魔幻数

网 络	魔 幻 数	十 六 进 制
主网络	0xD9B4BEF9	F9 BE B4 D9
测试网 3	0x0709110B	0B 11 09 07

另外，魔幻数常用于指示消息来源的网络。

与此同时，完整节点执行 4 项功能，其中包括钱包、矿工、区块链和网络路由节点。

当一个比特币核心节点启动时，首先初始化所有对等点的发现结果。该过程可通过查询 DNS 种子加以实现，如图 4.17 所示。其中，这一类种子被硬编码到比特币核心客户端，并由比特币社区成员维护。该查找最终返回一些 DNS A 记录。对于主网络，默认状态下，比特币协议在 TCP 端口 8333 上运行；对于测试网，则运行于 TCP 18333 端口上。

```
vSeeds.push_back(CDNSSeedData("bitcoin.sipa.be", "seed.bitcoin.sipa.be", true)); // Pieter Wuille
vSeeds.push_back(CDNSSeedData("bluematt.me", "dnsseed.bluematt.me")); // Matt Corallo
vSeeds.push_back(CDNSSeedData("dashjr.org", "dnsseed.bitcoin.dashjr.org")); // Luke Dashjr
vSeeds.push_back(CDNSSeedData("bitcoinstats.com", "seed.bitcoinstats.com")); // Christian Decker
vSeeds.push_back(CDNSSeedData("xf2.org", "bitseed.xf2.org")); // Jeff Garzik
vSeeds.push_back(CDNSSeedData("bitcoin.jonasschnelli.ch", "seed.bitcoin.jonasschnelli.ch", true)); // Jonas Schnelli
```

图 4.17 chainparams.cpp 中的 DNS 种子

首先，客户端发送协议消息 Version，其中包含了多个字段，如版本、服务、时间戳、网络地址、nonce 和其他一些字段。相应地，远程节点响应自身的版本消息，随后是两个节点之间的 verack 消息交换，这表明连接已经建立完毕。

在此之后，Getaddr 和 addr 消息彼此交换，以搜索客户端尚未发现的对等点。同时，

任何一个节点都可以发送一个 ping 消息查看连接是否依然存在。

至此，即可开始对区块进行下载。如果当前节点已经完全同步了所有区块，那么将使用 inv 协议消息监听新区块；否则，节点首先检查是否存在 inv 消息响应，并且是否已经存在存入清单。如果是，则使用 Getdata 协议消息请求区块；否则，使用 GetBlocks 消息请求存入清单。该方法一直被用至版本 0.9.3，对应过程如图 4.18 所示。

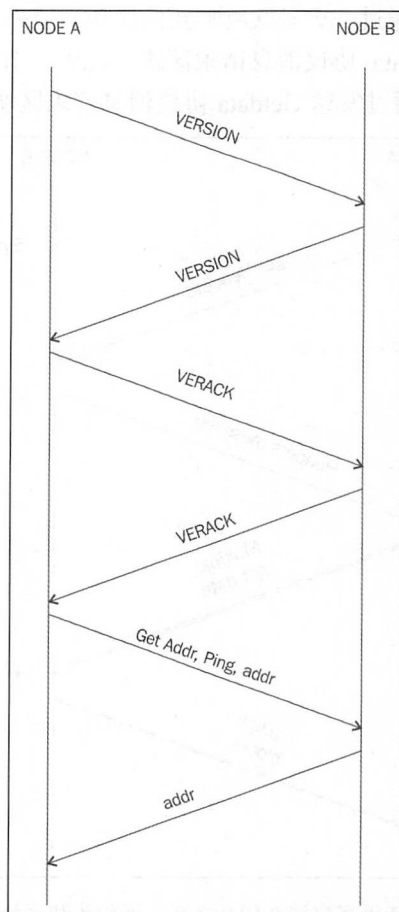


图 4.18 协议中，发现节点的可视化过程

根据比特币核心客户端的版本，初始区块的下载过程可采用区块优先或区块头优先的方法同步区块。其中，区块优先方法较为缓慢，自版本 0.10.0 之后便被弃用。

自版本 0.10.0 以来，首次引入了区块头优先的初始块下载方法，从而显著地改善了操作性能；而以前需要几天才能完成的区块链同步任务下载只需几个小时即可完成，其

核心思想可描述为，新节点首先查看节点头，并对其进行验证。一旦完成此任务，即可从现有的全部对等点中以并行方式请求区块，因为完整链的结构已经以区块头链的形式下载完毕。

在该方法中，如果头链已处于同步状态，客户端启动时将检查块链是否已经完全同步；否则，在客户端第一次启动时，将使用 `getHeader` 消息请求来自其他对等点的块头。如果块链是完全同步的，将通过 `inv` 消息侦听新的区块。另外，如果已经存在一个完全同步的区块头链，则使用 `GetData` 协议消息请求区块。此外，节点还检查消息头链是否比区块拥有更多的区块头，随后通过发送 `GetData` 协议消息请求区块。相关过程如图 4.19 所示。

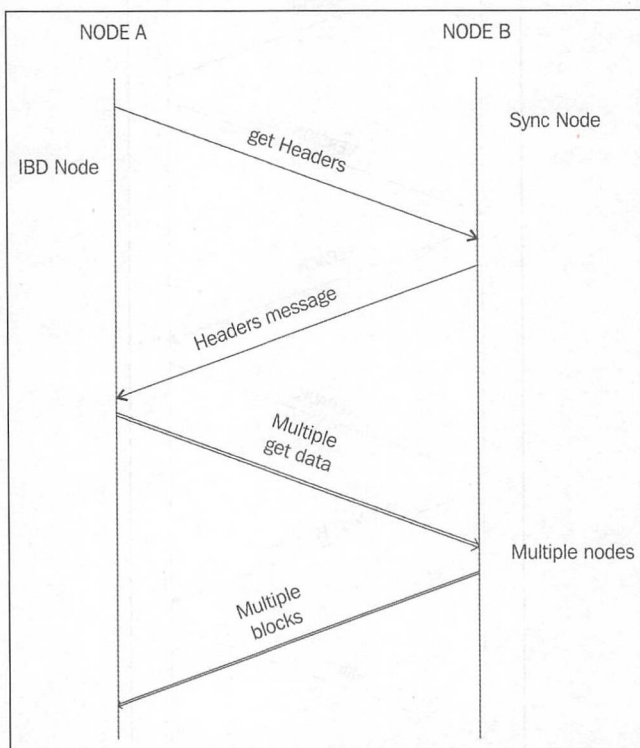


图 4.19 在比特币客户端 0.10.0 之后，区块头和区块的同步行为。

其中，IBD 表示初始区块下载；而节点表示区块所请求的节点

`Getblockchaininfo` 和 `Getpeerinfo` 这一类 RPC 新增了一些功能项，以迎合上述变化。其中，`getchaintips` 用于列出区块链的所有已知分支，且仅包括区块头；`Getblockchaininfo` 则用于提供与区块链当前状态相关的信息；而 `Getpeerinfo` 用于列出对等点之间公有的区块和区块头的数量。

Wireshark 还可以用于在对等点之间可视化消息交换过程，同时也是一种查看比特币协议的重要工具。作为一个基本的例子，图 4.20 显示了版本、verack、getaddr、ping、addr 和 inv 消息。

No.	Time	Source	Destination	Protocol	Length	Info
131	98.598526000	192.168.0.13	52.1.165.219	Bitcoin	192	version
150	99.188294000	192.168.0.13	52.1.165.219	Bitcoin	90	verack
151	99.188421000	192.168.0.13	52.1.165.219	Bitcoin	122	getaddr, ping
152	99.180715000	192.168.0.13	52.1.165.219	Bitcoin	1288	addr, getheaders[Malformed Packet]
486	112.053746000	192.168.0.13	52.1.165.219	Bitcoin	127	inv
818	143.630367000	192.168.0.13	52.1.165.219	Bitcoin	127	inv
1004	178.729768000	192.168.0.13	52.1.165.219	Bitcoin	127	inv

Transmission Control Protocol, Src Port: 52864 (52864), Dst Port: 18333 (18333), Seq: 207, Ack: 1291, Len: 1222						
Bitcoin protocol						
Packet magic: 0x0b110907						
Command name: addr						
Payload Length: 31						
Payload checksum: 0xa03fc07d						
Address message						
Count: 1						
Address: afbd025800ffff...						
Node services: 0x0000000000000000						
.....0 = Network node: Not set						
Node address: ::ffff:86.15.44.209 (::ffff:86.15.44.209)						
Node port: 18333						
Address timestamp: Oct 16, 2016 00:37:19.000000000 BST						
Bitcoin protocol						
Packet magic: 0x0b110907						
Command name: getheaders						
Payload Length: 1029						
Payload checksum: 0x4e54961d						
Getheaders message						
Count: 126						
Starting hash: 1101001f152142abccc039503abc56b149bd56c2b3925b65...						
Starting hash: 000000001980703bd53b0c7bf0ac995bccfeffd5cddc780...						
Starting hash: 000000007ad1fed813d20301b1762895a2e5b08c8a58b3ea...						
Starting hash: 000000003624c451f726a3e983d02279d9c7cf672d36f1d5...						

图 4.20 Wireshark 中的区块信息

具体而言，图 4.20 中可查看到包类型、命令行以及协议消息结果等重要信息。

这里显示了两个对等点之间数据流的协议图，以可以帮助对着理解何时启动节点，以及使用哪一种类型的消息。

在下面的例子中，比特币解析器将用于分析流量并识别比特币协议命令。其中可看到诸如版本、getaddr 和 getdata 等消息交换内容，并使用恰当的注释描述消息名称。这对于学习比特币非常有用，建议读者在比特币测试网络上进行尝试，并通过网络发送各种消息和事务，随后由 Wireshark 进行分析，如图 4.21 所示。

相应地，总共有 27 种协议消息类型，但随着协议内容的不断丰富，相关类型可能会随着时间的推移而增加。最常用的协议消息及其解释如下。

❑ version: 这是节点发送到网络的第一个消息，并发布其版本和区块计数。然后，

远程节点使用相同的信息进行响应，然后建立连接。

Time	192.168.0.13	136.243.139.96	Comment
97.734135000	(57868) →	version → (18333)	Bitcoin: version
98.025045000	(57868) →	verack → (18333)	Bitcoin: verack
98.025177000	(57868) →	getaddr.ping, addr → (18333)	Bitcoin: getaddr, ping, addr
98.025468000	(57868) →	getheaders → (18333)	Bitcoin: getheaders, [unknown command], [unknown command], [unknown command], headers
98.160419000	(57868) →	[TCP Retran] → (18333)	Bitcoin: [TCP Retransmission], getheaders, [unknown command], [unknown command], [unknown command]
98.598399000	(57868) →	getdata → (18333)	Bitcoin: getdata
144.343544000	(57868) →	inv → (18333)	Bitcoin: inv
176.152240000	(57868) →	getdata → (18333)	Bitcoin: getdata
179.493755000	(57868) →	getdata → (18333)	Bitcoin: getdata
218.101646000	(57868) →	ping → (18333)	Bitcoin: ping
218.192004000	(57868) →	[unknown.co] → (18333)	Bitcoin: [unknown command]
218.444431000	(57868) →	[TCP Retran] → (18333)	Bitcoin: [TCP Retransmission], [unknown command]
336.234936000	(57868) →	getdata → (18333)	Bitcoin: getdata
337.843423000	(57868) →	[unknown.co] → (18333)	Bitcoin: [unknown command]
338.143885000	(57868) →	ping → (18333)	Bitcoin: ping
448.764093000	(57868) →	getdata → (18333)	Bitcoin: getdata
457.894823000	(57868) →	[unknown.co] → (18333)	Bitcoin: [unknown command]
458.195265000	(57868) →	ping → (18333)	Bitcoin: ping
578.011774000	(57868) →	[unknown.co] → (18333)	Bitcoin: [unknown command]
578.212044000	(57868) →	ping → (18333)	Bitcoin: ping
585.587671000	(57868) →	inv → (18333)	Bitcoin: inv
647.169633000	(57868) →	inv → (18333)	Bitcoin: inv
671.962545000	(57868) →	getdata → (18333)	Bitcoin: getdata
698.037067000	(57868) →	[unknown.co] → (18333)	Bitcoin: [unknown command]
698.237350000	(57868) →	ping → (18333)	Bitcoin: ping
701.563581000	(57868) →	inv → (18333)	Bitcoin: inv
701.986269000	(57868) →	inv → (18333)	Bitcoin: inv
705.022173000	(57868) →	inv → (18333)	Bitcoin: inv
812.115878000	(57868) →	inv → (18333)	Bitcoin: inv
818.198570000	(57868) →	[unknown.co] → (18333)	Bitcoin: [unknown command]
818.298733000	(57868) →	ping → (18333)	Bitcoin: ping

图 4.21 消息交换示例

- ❑ verack: 接受连接请求时，表示版本消息的响应。
- ❑ inv: 由节点加以使用，进而通知区块和交易信息。
- ❑ getdata: 在请求哈希值标识的独立区块或交易时，表示基于 inv 的响应。
- ❑ getblocks: 返回一个包含全部区块(位于最后一个已知哈希值或 500 个区块之后)列表的 inv 包。
- ❑ getheaders: 用于特定范围内请求区块头。
- ❑ tx: 作为 getdata 协议消息的响应，用于发送某项交易。
- ❑ block: 发送区块，以响应 getdata 协议消息。
- ❑ headers: 该数据包将返回多达 2000 个区块头，作为对 getheader 请求的应答。
- ❑ getaddr: 这是作为获取已知对等点信息的请求发送的。
- ❑ addr: 提供了与网络节点相关的信息，以 IP 地址和端口号的形式包含了地址和地址列表。
- ❑ Full client and SPV client: 完整客户端是指，下载整个区块链的胖客户机或完整

的节点，这也是将区块链验证为客户端的最安全方法。比特币网络节点可以在两种基本模式下运行：完整客户端或轻量级 SPV 客户端。其中，SPV 客户端用于验证支付，且不需要下载完整的区块链。SPV 节点只保留当前最长区块链的区块头的副本。验证过的执行过程可描述为：查找将交易链接到原始区块（该区块接受当前交易）的 Merkle 分支，可以执行验证。相比较而言，BIP37 实现则更具实际意义，其中采用了 Bloom 过滤器仅过滤掉相关的交易事务。

- ❑ **Bloom filters:** Bloom 管理器基本上可表示为一种数据结构（带有索引的位向量），用于以概率的方式测试元素的成员，并提供了假阳性概率性查找（不包含假阴性）。通过多次哈希运算，随后通过索引将位向量中的相应位元设置为 1，元素即被添加到 Bloom 过滤器中。当检测 Bloom 过滤器中的元素时，可采用相同的哈希函数与位向量中的多个位相比较，以查看是否将相同的位设置为 1。需要注意的是，考虑到速度、不相关性以及均匀分布特征，并不是每一个哈希函数（例如 SHA1）都适用于 Bloom 过滤器。对于 Bloom 过滤器，最常用的哈希函数是 fnv、murmur 和 Jenkins。上述过滤器主要用于简单的支付验证 SPV 客户端，以请求交易及其感兴趣的 Merkle。其中，Merkle 块是区块的一个轻量级版本，包括区块头、哈希值、一个 1 位标记的列表和一个交易计数。随后，此类信息即可用来构建一棵 Merkle 树，该过程可通过创建一个过滤器予以实现，该过滤器只匹配 SPV 客户端请求的那些交易和区块。一旦版本消息被交换，并且在对等点之间建立了连接，节点就可以根据相关需求条件设置过滤器。这一类概率过滤器提供了不同程度的隐私性或精确度，取决于其精确程度或松散程度。严格的 Bloom 过滤器仅过滤被节点请求的交易，代价是将用户地址暴露给对方（将交易与其 IP 地址关联），从而损害用户的隐私。另外一方面，一个松散的设置表过滤器会检索更多不相关的交易，但隐私性将会得到提升。另外，对于 SPV 客户端，Bloom 过滤器允许它们使用低带宽，而非下载全部交易进行验证。
- ❑ **BIP 37:** 提出了比特币的实现方案，并向比特币协议引入了 3 条新消息。
 - **Filterload:** 用于在连接方面设置 Bloom 过滤器。
 - **Filteradd:** 向当前过滤器添加新的数据元素。
 - **FilterClear:** 删除当前加载的过滤器。

读者可参考 BIP 37 规范，以获取更多内容。

4.3.5 钱包

钱包软件用于储存私钥、公钥和比特币地址并执行各种功能，例如接收和发送比特

币。当前，该软件一般仅提供两种功能：比特币客户端和钱包。在磁盘上，比特币的核心客户端钱包软件存储为 Berkeley DB 文件，如下所示：

```
:~/Bitcoin$ file wallet.dat
```

其中，wallet.dat 表示 Berkeley DB（二叉树、版本 9、本地字节顺序）。

私钥可以采用不同的方式创建，并被不同类型的钱包所使用。注意，钱包不存储任何货币，同时也不存在用户存储余额或货币这一类概念。事实上，在比特币网络中，货币并不真实存在；相反，只有交易信息存储在区块链上（更准确地讲，是 UTXO，以及未消费的输出），随后用于计算比特币的数量。

1. 钱包类型

在比特币中，存在可以用来存储私钥的不同类型的钱包。作为一个软件程序，还可为用户提供某些功能来管理和执行比特币网络上的交易。

2. 非确定性钱包

这一类钱包中包含了随机生成的私钥，也被称为钥匙串钱包。比特币的核心客户端会在第一次启动时生成一些密钥；随后，还可在需要时生成密钥。管理大量的密钥是非常困难的，并且一项错误处理即可能导致货币的失窃或损失。此外，还需要创建密钥的常规备份并对其予以防护，以防止盗窃或丢失。

3. 确定性钱包

在这种类型的钱包中，密钥通过哈希函数生成种子值。其中，种子编号是随机生成的，通常由可读的助记码字表示。这里，助记码字是在 BIP39 中定义的，可以用来恢复所有密钥，并可降低私钥管理的难度。

4. 分层确定性钱包

HD 钱包定义于 BIP32 和 BIP44 中，并将源自种子的密钥存储于树结构中。该种子生成父密钥（主密钥），该密钥用于生成子密钥，然后是孙辈密钥。HD 钱包密钥构造方式并不会直接生成密钥；相反，这一过程仅生成一些信息（私钥生成信息），此类信息可以用来生成私钥序列。如果主私钥已知，HD 钱包中私钥的完整层次结构很容易恢复。正是因为这一特性，HD 钱包易于维护，而且便于携带。

5. 脑钱包

主私钥也可以从所记的密码哈希值中派生出来。此处的关键思想是，密码被用来派

生私钥。如果用于 HD 钱包，这将生成一个源自所记密码的、完整的 HD 钱包，这就是所谓的“脑钱包”。这种方法很容易遭受密码猜测和蛮力攻击，但是像钥延伸这一类技术可以用来减缓攻击者的进度。

6. 纸钱包

顾名思义，这是一种基于纸张的钱包，其上印有所需的关键材料。纸钱包在存储时需要一定的存储物理性。另外，纸钱包可通过在线方式从不同的服务提供商那里得到，例如 <http://bitcoinpaperwallet.com/> 或 <https://www.bitaddress.org/>。

7. 硬件钱包

另一种方法是使用一种防篡改设备来存储密钥。这种防干扰设备可以定制，或者伴随着 NFC 功能手机而出现，另外也可以是 NFC 手机中的安全元件(SE)。Trezor 和 Ledger 钱包（包含多种类型）则是最常用的比特币硬件钱包，如图 4.22 所示为 Trezor 钱包。



图 4.22 Trezor 钱包

8. 在线钱包

在线钱包具有在线存储功能，一般通过云服务予以实现，并为用户提供一个 Web 界面管理钱包，以执行各种功能，例如支付和收款。在线钱包容易使用，但用户须对在线钱包服务提供商予以足够的信任。

9. 手机钱包

手机钱包安装于移动设备上，可以提供各种支付方式，例如，使用智能手机摄像头快速扫描 QR 码并支付费用。Android 平台和 iOS 平台均支持手机钱包，如 breadwallet、copay 和 Jaxx，如图 4.23 所示为 Jaxx 手机钱包。

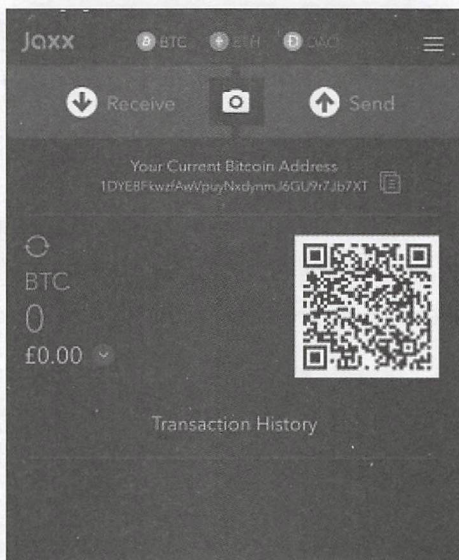


图 4.23 Jaxx 手机钱包

4.4 比特币支付

通过多种技术，比特币已逐渐成为一种可接受的支付方式。然而，在许多司法管辖区域内，比特币尚未成为合法货币。尽管如此，比特币越来越被许多在线商家和电子商务网站接受为支付方式。买家可以通过多种方式支付比特币。例如，在网上商店，可以使用比特币商家的解决方案，而在传统实体商店中，则可以使用销售终端和其他专用硬件。例如，客户可以利用卖家的支付 URI 简单地扫描二维码，并使用他们的移动设备付款。比特币可以通过点击链接或扫描二维码来支付。URI（统一资源标识符）基本上是表示交易信息的字符串，并在 BIP21 中加以定义。二维码可以显示附近的销售终端，几乎所有的比特币钱包都支持这一功能。

图 4.24 所示为 Logo 表示商家接受比特币支付。



图 4.24 接受比特币支付方式

另外,商业应用中也包含了各种支付方案,例如 xbtterminal 和 34 字节的比特币 POS 终端,如图 4.25 所示。



图 4.25 34 字节 POS 终端

许多在线服务提供商均提供了比特币支付处理器,并可与电子商务网站进行整合。对此,读者可简单地在互联网中进行搜索,即可发现诸多选项。

为了引入和规范比特币支付,目前已经涌现出多种 BIPs,其中最为引人注目的是 BIP 70。(安全支付协议)。BIP 70 制定了商家和客户之间安全通信的协议。该协议使用了 X.509 认证证书,并在 HTTP 和 HTTPS 上运行。该协议中涵盖了 3 种消息:PaymentRequest、Payment 和 PaymentACK。这一提议的主要特点是防御中间人攻击和安全支付证明。对于中间人攻击行为,攻击者位于商家和买家之间,而买家似乎是在与商家对话。但事实上,中间人正在与买家互动,而不是商家。因此,商家的比特币地址将被操控进而欺骗买家。

除此之外,一些其他的 BIP 也被提议,如 BIP71 和 BIP72,旨在标准化支付消息封装和 URI 方案,进而支持 BIP70。

4.4.1 比特币投资和比特币交易

一些在线交易平台推出了比特币交易服务,这也是互联网上的一项“大宗生意”。此类平台提供了比特币交易、CFD、点差交易、保证金交易和各种其他选项。当比特币的价格上涨或下跌时,交易者可以通过多头或空头仓位来购买比特币或交易。除此之外,许多在线比特币交易还提供了其他一些功能,例如其他虚拟货币的比特币交换、市场数

据、交易策略、图表，以及交易者所需的其他数据。图 4.26 展示了 CEX.IO 中的一个交易示例，其他交易平台也提供了类似的服务。

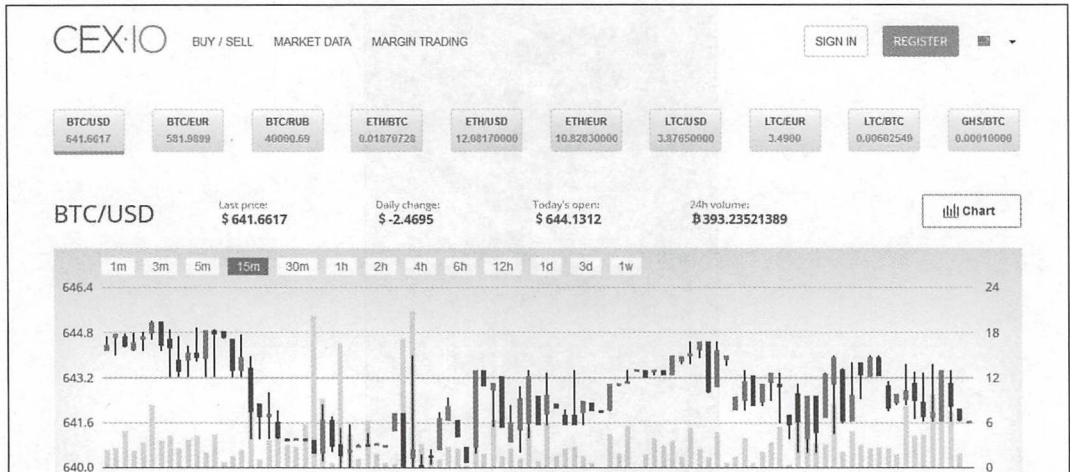


图 4.26 CEX.IO 中的比特币交易示例

图 4.27 则列出了交易过程中的全部订单。

Sell Orders			Buy Orders		
Total BTC available: 656.41831367			Total USD available: 390739.41		
Price per BTC	BTC Amount	Total: (USD)	Price per BTC	BTC Amount	Total: (USD)
642.4095	0.20450000	\$ 131.38	641.6210	0.01390000	\$ 8.92
642.4915	0.20910000	\$ 134.35	641.6201	0.23162780	\$ 148.62
643.4470	0.05000000	\$ 32.18	641.6200	0.12050000	\$ 77.32
643.4900	0.11944972	\$ 76.87	641.6117	1.83477084	\$ 1177.22
643.5000	1.85748652	\$ 1195.30	641.5584	0.30000000	\$ 192.47
643.6500	3.00000000	\$ 1930.95	641.5217	0.18180000	\$ 116.63
643.6999	0.13844181	\$ 89.12	641.0217	0.10000000	\$ 64.11
643.7000	45.80000000	\$ 29481.46	640.5300	0.67323160	\$ 431.23
643.7487	1.22995538	\$ 791.79	640.5000	0.40815400	\$ 261.43

图 4.27 CEX.IO 中的比特币交易订单

4.4.2 比特币安装

读者可访问 <https://bitcoin.org/en/download> 并安装比特币核心客户端。该客户端适用于 x86 Windows 以及 ARM Linux 下的不同架构和平台，如图 4.28 所示。

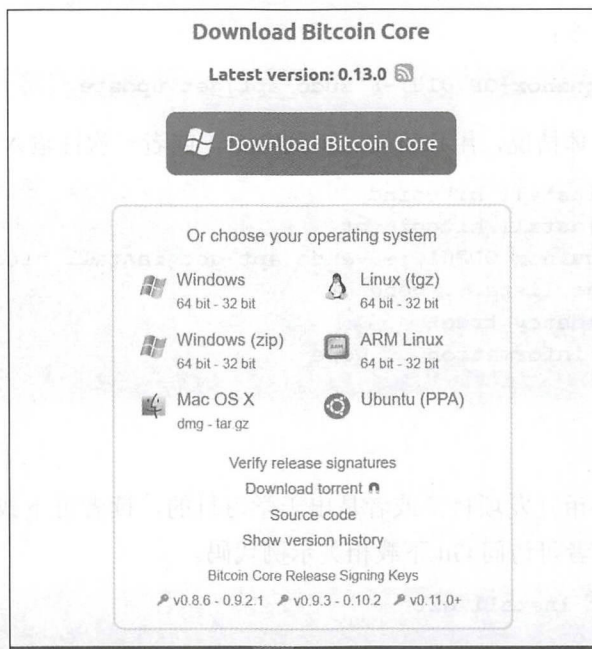


图 4.28 安装比特币客户端

1. 设置比特币节点

此处展示了 Ubuntu 中的比特币运行示例;对于其他平台,读者可访问 www.bitcoin.org 获取详细信息。运行结果如图 4.29 所示。

```
drequinox@drequinox-OP7010: ~  
drequinox@drequinox-OP7010:~$ sudo apt-add-repository ppa:bitcoin/bitcoin  
[sudo] password for drequinox:  
Stable Channel of bitcoin-qt and bitcoind for Ubuntu, and their dependencies  
More info: https://launchpad.net/~bitcoin/+archive/ubuntu/bitcoin  
Press [ENTER] to continue or ctrl-c to cancel adding it  
  
gpg: keyring '/tmp/tmpzsl4ltrx/secring.gpg' created  
gpg: keyring '/tmp/tmpzsl4ltrx/pubring.gpg' created  
gpg: requesting key 8842CE5E from hkp server keyserver.ubuntu.com  
gpg: /tmp/tmpzsl4ltrx/trustdb.gpg: trustdb created  
gpg: key 8842CE5E: public key "Launchpad PPA for Bitcoin" imported  
gpg: no ultimately trusted keys found  
gpg: Total number processed: 1  
gpg:         imported: 1 (RSA: 1)  
OK  
drequinox@drequinox-OP7010:~$
```

图 4.29 Ubuntu 中的比特币运行示例



随后输入下列命令：

```
drequinox@drequinox-OP7010:~$ sudo apt-get update
```

根据客户端的具体情况，用户可输入下列命令，或者一次性输入两种命令：

```
sudo apt-get install bitcoind
sudo apt-get install bitcoin-qt
drequinox@drequinox-OP7010:~$ sudo apt-get install bitcoin-qt bitcoind
Reading package lists... Done
Building dependency tree
Reading state information... Done
.....
```

2. 编写源代码

无论是参与比特币开发项目，或者是出于学习目的，读者可下载相关源代码并对其进行编译。对此，读者可访问 Git 下载相关示例代码。

```
$ sudo apt-get install git
$ mkdir bcsourse
$ cd bcsourse
drequinox@drequinox-OP7010:~/bcsourse $ git clone
https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 78960, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 78960 (delta 0), reused 0 (delta 0), pack-reused 78957
Receiving objects: 100% (78960/78960), 72.53 MiB | 1.85 MiB/s, done.
Resolving deltas: 100% (57908/57908), done.
Checking connectivity... done.
drequinox@drequinox-OP7010:~/bcsourse$
```

将其目录修改为 bitcoin，如下所示：

```
drequinox@drequinox-OP7010:~/bcsourse$ cd bitcoin
```

上述各项步骤处理完毕后，即可对其进行编译，如下所示：

```
drequinox@drequinox-OP7010:~/bcsourse/bitcoin$ ./autogen.sh
drequinox@drequinox-OP7010:~/bcsourse/bitcoin$ ./configure.sh
drequinox@drequinox-OP7010:~/bcsourse/bitcoin$ make
drequinox@drequinox-OP7010:~/bcsourse/bitcoin$ sudo make install
```



3. 设置 bitcoin.conf

bitcoin.conf 文件是比特币核心客户端用来保存配置设置的配置文件。除了可以在配置文件中设置 -conf 开关之外, 比特币客户端的所有命令行选项均可在配置文件中设置, 当启用 bitcoin -qt 或 bitcoind 时, 将从该文件中获取配置信息。

在 Linux 系统中, 对应文件位于 \$HOME/.bitcoin/ 中; 或者也可以在命令行中使用 -conf=<file> 转换到 bitcoind 核心客户端软件。

4. 在测试网中启动一个节点

如果用户希望对比特币网络进行测试, 并尝试运行一个实验项目, 那么可以在测试网 (testnet) 模式下启动比特币节点。与 Live 网络相比, 测试网络的速度更快, 并且对挖掘和交易设置了较为宽松的规则。

相应地, 存在多种服务可以用于比特币测试网络, 例如比特币 TestNet 沙箱, 用户可向测试网比特币地址请求所支付的比特币。对此, 用户可以访问 <https://test.manu.backend.hamburg/>, 这对于测试网络上的交易试验非常有用。

启动测试网的对应命令行如下所示:

```
bitcoind --testnet -daemon
bitcoin-cli --testnet <command>
bitcoin-qt --testnet
```

5. 启动回归测试节点

出于测试目的, 回归测试模式可用于构造本地区块链。

针对于此, 下列命令可启动回归测试模式下的一个节点:

```
bitcoind -regtest -daemon
Bitcoin server starting
```

区块则通过下列命令构建:

```
bitcoin-cli -regtest generate 200
```

在 Linux 系统中, 日志文件则可在 debug.log 的 .bitcoin/regtest 目录下查看, 如图 4.30 所示。

```
drequinox@requinox-KT010:~/bitcoin/regtest$ tail -f debug.log
2016-10-16 15:43:55 AddToWallet d461e1f2c4d5695139e2aebef9993fcd551b1a4e3a8e5b77e477cd90dd6a new
2016-10-16 15:43:55 CreateNewBlock(): total size 1000 txs: 0 fees: 0 sigops 400
2016-10-16 15:43:55 UpdateTip: new best=37e1f40239a3724d6edf43d26955eb3508b5f27405289ef9204e53fe4eb87 height=299 version=0a30000003 log2_work=9.22881
57 tx=300 date='2016-10-16 15:44:27' progress=1.000000 cache=0.1MiB(298tx)
2016-10-16 15:43:55 AddToWallet b082cbe122c4f3aee4b53e4026d3fae0916c570df2b154fabf51c6767f9d70ef new
2016-10-16 15:43:55 CreateNewBlock(): total size 1000 txs: 0 fees: 0 sigops 400
2016-10-16 15:43:55 UpdateTip: new best=5c2220b0306af3f4978fbb14803d1d34ec0fb697a199d302bebd88de43ad2 height=300 version=0a30000003 log2_work=9.23361
97 tx=301 date='2016-10-16 15:44:28' progress=1.000000 cache=0.1MiB(300tx)
2016-10-16 15:43:55 AddToWallet c315c5b6863aed234477f6ee6c0db7ace273f10549d249690b8793de0de0b8e1 new
2016-10-16 15:43:55 CreateNewBlock(): total size 1000 txs: 0 fees: 0 sigops 400
2016-10-16 15:43:55 UpdateTip: new best=7f9eeb77e0d344784426c95aeb32c03b10715374c0a87ec93118ab77ae8f8ae height=301 version=0a30000003 log2_work=9.23850
41 tx=302 date='2016-10-16 15:44:28' progress=1.000000 cache=0.1MiB(301tx)
2016-10-16 15:43:55 AddToWallet 420058e9e73f6602f3e17c999efa4062ad2643b255630d7e2ec086e7f5ac029 new
```

图 4.30 日志文件



待区块构建完毕后，即可通过下列命令进行查看：

```
drequinox@drequinox-OP7010:~/bitcoin/regtest$ bitcoin-cli -regtest  
getbalance  
8750.00000000
```

下列命令可用于终止某一个节点：

```
drequinox@drequinox-OP7010:~/bitcoin$ bitcoin-cli -regtest stop  
Bitcoin server stopping
```

6. 在 Live 主干网中启动节点

Bitcoin 是可以作为后台程序运行的核心客户端软件，同时提供了 JSON RPC 接口。

bitcoin-cli 是一个功能丰富的命令行工具，可以与后台程序交互；随后，后台进程与区块链交互并执行各项功能。除此之外，bitcoin-cli 只调用 JSON-RPC 函数，并且在区块链中不执行任何操作。

bitcoin-qt 是比特币核心客户端 GUI。当首先启动钱包软件时，将验证磁盘上的区块，随后启动并显示如图 4.31 所示的 GUI。

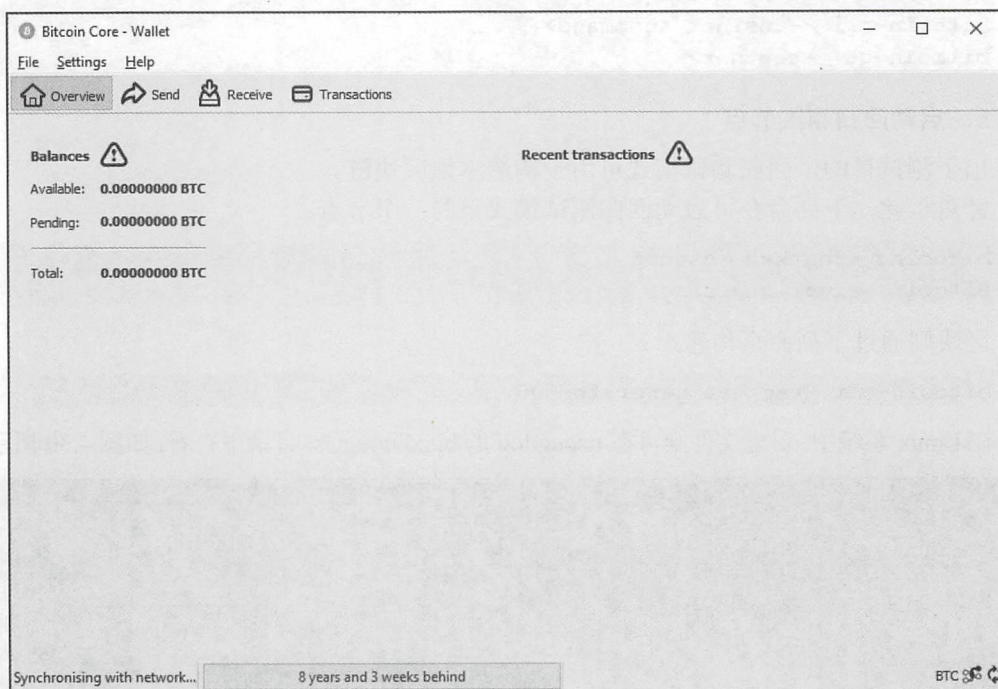


图 4.31 Bitcoin Core QT 客户端。安装刚刚完成后，区块链并未处于同步状态



验证处理过程并非仅特定于 Bitcoin-qt 客户端，也可由 bitcoind 客户端执行。

7. bitcoin-cli 试验

bitcoin-cli 是比特币核心客户端可用的命令行接口，通过比特币核心客户端提供的 RPC，可用于执行各种功能，如图 4.32 所示。

```
drequinox@drequinox-OP7010:~$ bitcoin-cli getinfo
{
  "version": 130000,
  "protocolversion": 70014,
  "walletversion": 130000,
  "balance": 0.00000000,
  "blocks": 433948,
  "timeoffset": 0,
  "connections": 8,
  "proxy": "",
  "difficulty": 258522748404.5154,
  "testnet": false,
  "keypoololdest": 1475534258,
  "keypoolsize": 100,
  "paytxfee": 0.00000000,
  "relayfee": 0.00001000,
  "errors": ""
}
drequinox@drequinox-OP7010:~$
```

图 4.32 bitcoin-cli getinfo 运行示例。可采用相同的格式调用其他命令

通过如图 4.33 所示的命令，可显示全部命令列表。

```
drequinox@drequinox-OP7010:~$ bitcoin-cli -testnet help | more
== Blockchain ==
getbestblockhash
getblock "hash" ( verbose )
getblockchaininfo
getblockcount
getblockhash index
getblockheader "hash" ( verbose )
getchaintips
getdifficulty
getmempoolancestors txid (verbose)
getmempooldescendants txid (verbose)
getmempoolentry txid
getmempoolinfo
getrawmempool ( verbose )
gettxout "txid" n ( includemempool )
gettxoutproof ["txid",...] ( blockhash )
gettxoutsetinfo
verifychain ( checklevel numblocks )
verifytxoutproof "proof"

== Control ==
getinfo
help ( "command" )
stop
```

图 4.33 测试网络中的 bitcoin-cli。限于篇幅，图中仅显示了部分命令

自比特币核心客户端 0.10.0 开始，还可以使用 HTTP REST 接口。默认情况下，这将



运行在与 JSON-RPC 相同的 TCP 端口上，即 TCP 端口 8332。

4.4.3 比特币编程和命令行接口

比特币编程涵盖了十分丰富的话题。其中，比特币核心客户端包含各种 JSON RPC 命令，并可用于构造原始交易事务，并通过定制脚本或程序执行其他功能。此外，还可以使用命令行工具 bitcoin-cli，它使用了 JSON-RPC 接口，并提供了可与比特币协同工作的、丰富的工具集。

此类 API 还可通过比特币 API 的形式，由在线服务提供商提供，包括简单的 HTTP REST 接口。比特币 API 提供了大量的选项，用户可据此开发基于比特币的解决方案，例如 blockchain.info、bitpay 和 block.io。

比特币编程涉及大量的库，如下所示。

- ❑ Libbitcoin: 读者可访问 <https://libbitcoin.dyne.org/>，获取功能强大的命令行工具和客户端。
 - ❑ Pycoin: 一个 Python 库，读者可访问 <https://github.com/richardkiss/pycoin> 下载。
 - ❑ Bitcoinj: 该库以 Java 实现，读者可访问 <https://bitcoinj.github.io/> 进行下载。
- 除此之外，还存在大量的在线比特币 API，其中较为常用的 API 包括以下方面：
- ❑ <https://bitcore.io/>。
 - ❑ <https://bitcoinjs.org/>。
 - ❑ <https://blockchain.info/api>。

基本上讲，上述 API 具有相似的功能，且难以选出最优方案。

4.4.4 比特币改进协议（BIP）

这一类文档用于通知比特币社区，相关内容包括所建议的改进措施、设计问题，以及与比特币生态系统相关的信息。比特币改进协议的缩写为 BIP，并涵盖了下列 3 种类型。

- ❑ 标准 BIP: 用于描述对比特币系统产生重大影响的主要变化，例如，区块大小更改、网络协议更改或交易验证的更改。
- ❑ 流程 BIP: 标准 BIP 和流程 BIP 之间的一个主要区别是，标准 BIP 涉及协议的改变；而流程 BIP 通常处理的是核心比特币协议之外的流程的变化，这些都是在比特币用户达成共识后才实施的。
- ❑ 信息 BIP: 通常用于建议或记录一些与比特币生态系统相关的信息，例如设计问题。



4.5 本章小结

本章介绍了比特币及其相关概念，涉及与比特币的历史发展和基本定义；随后引入了密钥、地址以及公钥和私钥等概念。除此之外，本章还讨论了比特币网络中交易的工作方式（以及相关的概念，如脚本、操作码和交易类型）。同时，本章还讲解了支持比特币网络的区块链。此外，其他内容还包括采矿、工作量证明、采矿挖掘系统和钱包等概念。最后，本章还提供了一些设置比特币客户端的实用信息、`bitcoin-cli` 应用，以及不同类型的比特币网络。第5章将介绍替代货币和相关概念。



第5章 替代币

比特币获得巨大成功随后一些替代币（山寨币）项目即启动。比特币于 2009 年发布，2011 年便推出了第一个替代币项目（名为 Namecoin）。在 2013 年和 2014 年，替代币市场呈指数增长，其发展也是良莠不齐。根据替代币发展的主要目的，可以将其大致分为两类。如果主要目的是建立一个去中心化的区块链平台，则称为替代链；如果替代项目的唯一目的是引入一种新的虚拟货币，则称作替代币。替代区块链将在本书后续章节中详细讨论。

这一章主要介绍替代币（altcoin），旨在引入一种新的虚拟货币。尽管某些人提出了基于比特币替代协议，并以此提供各项服务，例如 Namecoin，但其主要目的是提供去中心化的命名和身份服务，而不是货币。

截至 2016 年年底，市场上出现了数百种替代币，且具有一定的货币价值。其中，大量替代项目均是比特币源代码的直接分支，仅少量为原创项目。相应地，一些替代币开始着手解决比特币中的限制条件，比如隐私。另一些替代币则提供不同类型的挖掘机制、阻塞时间调整和分配方案。

根据定义，在产生硬分叉的情况下即可产生一种替代币。如果比特币包含了一个硬分叉，则旧链可视作一种替代币。然而此处并不存在明文规定以表明哪一个链将变为替代币。这种情况近期出现于以太坊（Ethereum, ETH）中。除了以太坊货币之外，硬分叉导致一种新型货币——ETC（即以太坊经典）的产生。以太坊经典表示为旧链，而 Ether 则是分叉后的新链。这里的硬分叉引发了诸多争议，其中也涉及了多种原因。首先，这违背了去中心化的核心精神。作为中心实体，以太坊基金会决定继续维持这一局面，尽管并非每个人都拥护这一主张。其次，鉴于在硬分叉方面的分歧，这一决定也会对拥护群体造成分裂。虽然硬分叉在理论上构造了一种替代币，但也存在一定的局限性：即使变化导致了硬分叉的出现，但货币的基本参数并未发生显著变化，且通常保持不变。基于这一原因，可重新开始编写一种新型货币，或者利用期望的参数和特性，通过比特币分叉（或另一种货币的源代码）构造一种新货币。

替代币必须能够吸引新的用户、交易和矿工，否则货币将不存在任何价值。鉴于网络效应和社区的可接受性，货币可具有一定的价值，特别是在虚拟货币空间中。如果一种货币无法吸引足够多的用户，那么它很快就会被遗忘。针对于此，可通过提供初始数量的货币（及其相关方法）来吸引用户，其中包括以下内容。



- ❑ 构造新的区块链：替代币可以创建一种新的区块链，并将货币分配给矿工。但随着多起诈骗事件的出现，矿工在启动一种新货币并获利后随即消失。因此，这种方法现在已变得不受欢迎。
- ❑ “燃烧”证明：另一种将初始资金分配给新的替代币的方法是燃烧证明，也称为单向限定或价格上限。在这种方法中，用户永久地销毁一定数量的比特币，与所发行的替代币数量成比例。例如，如果 10 个比特币被毁，那么替代币的价值就不会超过所销毁的比特币数量。这基本上意味着比特币通过燃烧被转换成替代币。
- ❑ 所有权证明：还有一种方法是证明用户拥有一定数量的比特币，而不是永久地销毁比特币。通过将替代币区块链接至比特币区块，所有权证明可用于发行替代币。例如，这可以通过合并采矿来实现，在这种情况下，比特币矿工可以在没有任何额外工作的情况下挖掘替代币区块，同时挖掘比特币。
- ❑ 楔入式侧链技术：顾名思义，侧链是与比特币网络分离的区块链，但比特币可以转移至侧链。同样，替代币也可以被转移回比特币网络，这一概念叫作双向楔入（peg）。

同样，替代币的投资和交易也是一项大宗“生意”，尽管其规模与比特币相比稍逊一筹，但也足以吸引新的投资者和交易人员，并为市场提供流动性。图 5.1 显示了 <http://coinmarketcap.com> 中提供的替代币市值总和。

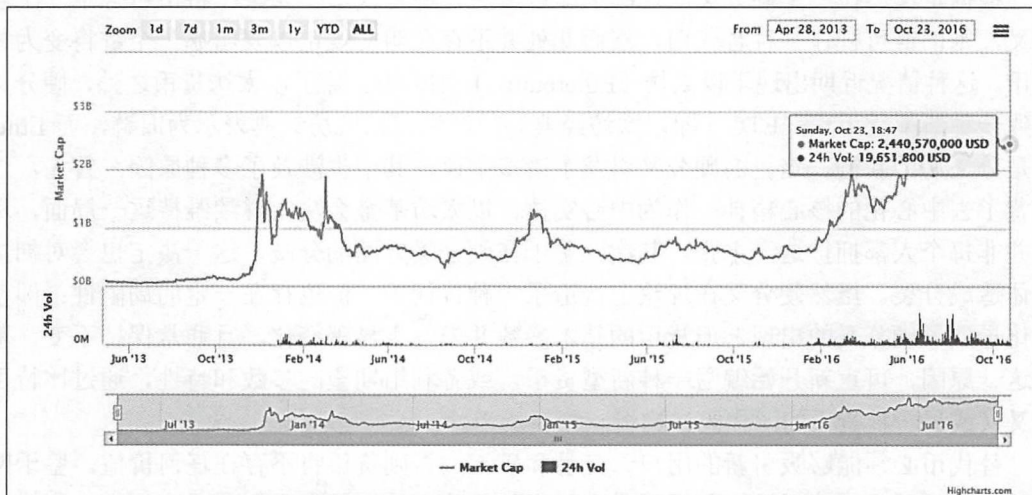


图 5.1 在本书编写时，替代币市场市值总和为 20 亿美元

图 5.2 显示了目前市值最高的 10 种货币。



Name	Market Cap	Price
Bitcoin	£8,983,068,268	£563.40
Ethereum	£806,208,216	£9.44
Ripple	£248,473,625	£0.01
Litecoin	£159,226,903	£3.31
Ethereum Classic	£69,366,041	£0.81
Monero	£67,664,027	£5.12
Dash	£54,675,636	£8.00
Augur	£45,221,626	£4.11
MaidSafeCoin	£30,929,833	£0.07
NEM	£28,761,896	£0.00

图 5.2 数据源自 <https://coinmarketcap.com/>

替代币的出现涉及多种因素，同时也引入了一些全新的概念。一些概念甚至先于比特币。比特币首次引入了拜占庭将军问题这一类新概念，同时，哈希现金和工作量证明也通过巧妙的方式被予以运用，并成为人们关注的焦点。随着替代币项目的出现，各种新的技术和概念也层出不穷。为了更好地理解当前替代加密货币的现状，首先应了解一些基础理论知识。下面将介绍一些与替代币项目相关的新概念。

5.1 理论基础

在过去几年，随着替代币的出现，各种理论性概念也应运而生。

5.1.1 工作量证明的替代方案

在密码学背景下，“工作量证明”（PoW）方案最初用于比特币中，并提供了一种保证机制，即矿工完成了所需数量的工作并找到一个区块。这为区块链提供了去中心化、安全性和稳定性。此外，PoW 也是比特币的主要载体，提供了去中心化的分布式共识机制。PoW 方案应具备一种称作“进度自由”（progress freeness）的特性，也就是说，消耗计算资源的奖励应具有随机性，并与矿工做出的贡献成比例。在这种情况下，即使是那些能力相对较弱的矿工，也有可能获得奖励。“进度自由”这一术语是 Arvind Narayanan 等人在其著作 *Bitcoin and Cryptocurrency Technologies* 中提出的。对于挖掘计算谜题，其他要求还包括可调难度和快速验证。其中，可调难度确保区块链上的挖掘难度，进而响

应不断增加的哈希计算强度和用户数量。作为一种属性，快速验证意味着挖掘计算谜题应体现一定的简单性和快捷性。除此之外，PoW 方案还引入了 ASIC，尤其是在比特币（Double SHA-256）应用中。其间，计算正转向那些能够负担得起大规模 ASIC 的矿工或矿池，这也对比特币去中心化的核心理念提出了挑战。

除此之外，还存在一些备选方案，例如抗 ASIC 谜题，其设计理念可描述为：针对此类谜题构造 ASIC，且不会在商用硬件上提升计算性能。针对这一目的，一种较为常见的技术则是内存计算谜题。这种方法的核心思想是，由于求解谜题需要占用大量的内存，所以在基于 ASIC 的系统上实现是不可行的。这种技术最初运用于莱特币和 Tenebrix 中，其中 Script 哈希函数被用作抗 ASIC PoW 方案。尽管这一方案最初生成具有抗 ASIC 特征，但最近的 Script ASIC 已经逐渐违背了莱特币的设计初衷。

抗 ASIC 的另一种方法是，需要计算多个哈希函数以提供 PoW，这也称为链式哈希方案。该方案背后的原理是，在 ASIC 上设计多个哈希函数缺乏可行性。最常见的例子是在 Dash 中实现 X11 内存硬函数。这里，X11 由 11 名 SHA3 竞争者组成，其中，某一算法将计算后的哈希结果输出到下一个算法，直到所有 11 种算法以序列方式使用。这一类算法包括 blake、bmw、groestl、jh、keccak、skein、luffa、cubehash、shavite、simd 和 echo。

最初，上述方法提供了抗 ASIC 性；当前 ASIC 矿工可以在商业上获得支持，同时支持 X11 和类似的方案。最近的一个例子是 ASIC Baikal Miner，并支持 X11、X13、X14 和 X15 挖掘。其他例子还包括 iBeLink DM384M X11 矿工和 Pinidea X11 ASIC 矿工。

另一种方法则是自突变谜题的设计问题。该问题可智能或随机地改变 PoW 方案，或者将其需求条件作为时间的函数。考虑到需要为每项功能设计多个 ASIC，因而其难以在 ASIC 中实现。另外，随机调整方案也难以在 ASIC 中加以处理。当前，其实现方式尚不得知。

PoW 方案的最大缺点是能耗问题。据估计，到 2020 年，比特币矿工的总耗电量将与丹麦全国的耗电量相当。这一数字无疑是巨大的，因而节能问题也提到具体日程上来。事实上，大部分电力均消耗在采矿上。环保人士也对这种情况表示了担忧。

曾有人提出，PoW 方案可以设计成两种用途。首先，其主要目的是达成一致的共识机制；其次是执行有效的科学计算。这种方法不仅可以用于采矿，还解决其他科学问题。此类工作量证明最近被素数币（Primecoin）实现，其中需要获取特定的素数链，例如 Cunningham 链和 bi-twin 链。素数分布学科（例如物理学）在研究中具有特殊意义，不仅可使矿工获得区块奖励，而且有助于寻找特定的素数。

1. 存储证明

存储证明也被称作不可恢复证明，这是另一种有用的工作量证明，且需要存储大量

的数据。该方案由微软研究院发布，为归档数据的分布式存储提供了有益的帮助。矿工被要求存储一个伪随机选取的大规模数据子集以执行采矿。

2. 权益证明

权益证明也称作虚拟挖掘，可视为另一种挖掘谜题，也可作为传统的 PoW 方案的替代方案。2012 年 8 月，该方案最初在 PeerCoin 中提出。其中，用户需要证明拥有一定数量的货币，从而证明其货币权益。

其中，最简单的权益关系可描述为，对于那些拥有大量数字货币的用户来说，挖掘过程相对容易。该方案的益处包含两个方面：首先，与买入高端 ASIC 设备相比，获取大量的数字货币相对困难，其次是可节省计算资源。下面将对各种权益形式进行逐一讨论。

- ❑ 货币证明。币龄是货币最后使用或被持有的时间。这是一种不同于常规权益证明形式的方法，即在替代币中拥有最高份额的用户可以更容易地进行采矿。在基于币龄的方法中，货币的年龄在每次开采时都被重置。对于一段时间内持有且未消费货币的矿工，则会受到一定的奖励。当与 PoW 结合使用后，该机制通过一种创新方式实现于 Peercoin 中。另外，采矿谜题的难度与币龄成反比，也就是说，如果矿工使用货币-份额交易消费币龄，那么 PoW 的需求条件就会得到一定的缓解。
- ❑ 储蓄证明。这一方案背后的核心理念是，矿工新开采的区块在一定时间内无法使用。更确切地说，在采矿作业期间，货币会被锁定在一定数量的区块上。该方案的工作原理是，以一定时间内冻结一定数量的货币为代价，以使矿工执行采矿操作。这也是权益证明的一种类型。
- ❑ 燃烧证明。作为一种计算能力的替代支出，燃烧证明实际上会破坏一定数量的比特币，以获得等价的替代币。在启动新货币项目时，这通常可提供相对公平的初次分配。这也可视作是一种替代的采矿方案，其中新货币的价值源自以前一定数量的硬币被销毁这一事实。
- ❑ 行动证明。该方案是 PoW 和权益证明的一种混合形式，在该方案中，区块最初由 PoW 生成，但随后每个区块随机分配 3 名权益相关者，并需要对其进行数字签名。后续块的有效性依赖于之前随机选取区块的有效签名。

然而，此处仍存在一种零风险问题，即创建一个区块链的分支将不再重要——这类情况是可能出现的，因为在 PoW 中需要适当的计算资源进行挖掘，而在权益证明中则无此类要求。因此，攻击者可以使用同一货币在多个链上进行挖掘。

3. 非外包谜题

这一问题背后的主要思想是阻止矿池的扩展。如前所述，矿池向所有参与者提供奖励，并与所消耗的计算能力成正比。然而，在该模型中，矿池操作者表示为中心权威机构，负责奖励的分配并制订特定的规则。除此之外，矿工具有共同目标（从矿池管理者处获取奖励），因而全部矿工均彼此信任。相比之下，非外包谜题则允许矿工自行获取奖励。因此，鉴于匿名矿工之间的内在不信任性，难以形成有效的矿池。

5.1.2 难度调整和目标重定位算法

另一个概念则是重定位算法中的难度问题，该问题是随着比特币和替代币的出现而产生的。在比特币中，难度目标可简单地通过下列公式进行计算；然而，其他货币也形成了自己的算法，或者实现为比特币难度算法的修正版本。

$$T = \text{上次时间} \times \text{实际时间} / 2016 \times 10 \text{ 分钟}$$

比特币难度控制背后的思想是，生成 2016 个区块大约需要两周时间（区块间约为 10 分钟）。如果 2016 个区块的挖掘时间超出两周，难度将会递减；若小于两周，难度则增加。当引入 ASIC 时，考虑到较高的区块生成率，难度则呈指数级增加，这也是非 ASIC 抗性的 PoW 算法的缺陷之一，并导致挖掘能力过于集中化。除此之外，另一个问题则体现在：如果启动某一新型货币，并采用与比特币相同的工作量证明（基于 SHA256），那么恶意用户仅需使用 ASIC 矿工即可控制全部网络。如果对这一新型货币缺少足够的兴趣，或者某人通过消耗较高的计算资源掌控网络，那么这一攻击将变得更具实际意义。但如果包含类似计算能力的矿工也加入了这一替代币网络，那么该攻击行为一般缺乏可行性——矿工将处于彼此竞争状态。此外，多个矿池可能会造成更大的威胁，其中，一组矿工可以自动转换至有利可图的货币。这种现象被称为“池跳跃”（pool hopping），并对区块链产生负面影响，从而导致替代币的增长。“池跳跃”现象会对网络产生不利影响，因为只有在难度很低时，跳跃者才会加入到网络中，并可以迅速获得奖励，当难度上升（或重新调整）时离开，随后当难度调整回来时重现。例如，当快速开采某一种新型货币时，如果某一多池（multipool）在消耗其资源，那么难度将会很快增加；当多池离开货币网络时，将变得无法使用，其原因在于，现有难度已经增加到这样一种水平：独立矿工已不再盈利，因而无法继续维持。解决这一问题的唯一方法是启动一个硬分叉，该分叉通常不会受到社区的欢迎。

稍后将讨论相关算法以解决此类问题。所有这些算法其思想基于重新调整各种参数，以响应哈希速率变化。相关参数包括前区块的数量、前区块的难度、调整的比例，以及

被重新调整的难度值。在接下来的章节中，将针对替代币介绍几种不同的难度算法。

1. Kimoto 重力井 (KGW)

该算法适用于各种不同的替代币，用于调节难度。Kimoto 重力井首次出现于美卡币中，采用自适应方式调整每个区块的网络难度。该算法的逻辑如下：

$$KGW = 1 + (0.7084 \times \text{pow}((\text{double}(\text{PastBlocksMass}) / \text{double}(144)), -1.228))$$

基本上讲，该算法运行在一个循环结构中，通过一组预先确定的区块 (PastBlockMass) 计算一个新的调整值。其核心思想是开发一种自适应的难度调节机制，进而重新调整哈希速率峰值时的难度。Kimoto 重力井确保在区块之间的时间保持大致相同。在比特币中，难度每隔 2016 个区块即会调整一次；但在 KGW 中，难度会在每个区块进行调整。

该算法容易受到时间错位 (time warp) 攻击，攻击者可以在创建新区块时暂时面临较低的难度。另外，攻击过程中存在一个时间窗口，在这个窗口中难度将有所降低，攻击者可快速生成大量硬币。

2. 暗黑重力波

暗黑重力波 (DGW) 是一种新的算法，用于解决在 KGW 算法中出现的时间错位问题。DGW 首次出现于达世币中，之前被称为暗黑币。该算法利用多指数移动平均值和简单的移动平均值实现更平滑的调整机制。对应公式如下：

$$2222222 / ((\text{难度} + 2600) / 9)^2$$

作为一种难度调整机制，该公式实现于达世币和其他替代币中。

DGW 3.0 版本是该算法的最新实现，与 KGW 相比，该算法支持改进后的重定向操作。

3. DigiShield

DigiShield 是另一个难度重定向算法，稍作修改并经历了多次试验后，现用于 Zcash 中。该算法考察固定数量的前区块，并计算生成时间；随后将实际时间间隔除以平均目标时间，进而将对其难度调整至前区块的难度。在该方案中，重定向的计算速度显著提升，并可快速从哈希速率的突发性增长或降低中恢复。除此之外，该算法还可防止多池现象，从而引发哈希速率的快速增加。根据具体实现，网络难度按照每个块或每分钟重新调整。与 KGW 相比，该算法的创新之处在于快速地调整时间。

4. MIDAS

与上述算法相比，多区间难度调整系统 (MIDAS) 相对复杂，该算法对于哈希速率的突然变化反应更加迅速。另外，该算法还对时间错位攻击提供了保护。

大量算法的出现旨在解决比特币中的各种限制条件，下面将对此加以讨论。

5.2 比特币中的限制条件

比特币中的各种限制条件也引发了对替代币的关注，相关方法针对比特币中的各种限制提供了解决方案。其中，最为知名且被广泛讨论的内容则是比特币的匿名性。

5.2.1 隐私和匿名性

由于区块链定义为所有交易的公共账本，而且具有公开特征，因此对其进行分析并无太多必要。当与流量分析相结合时，交易可以链接回源 IP 地址，从而可能显示交易的发起者。从隐私的角度来看，这一问题应引起足够的重视。在比特币中，针对每项交易生成一个新地址是一类较为常见的做法，因而也支持一定程度上的非相关性（不可链接性）。除此之外，各种技术也用于跟踪网络中的交易流，并可将其链接回发起者。研究人员采用各种方法分析诸如事务图、地址图和实体图等区块链，从而将用户与交易联系起来。相应地，该过程也引发了隐私问题。对此，前面提到的分析技术可以得到进一步的扩展，即使用与交易相关的公开信息，并将其链接至实际用户。此处，可以使用开源区块解析器来从区块链数据库中提取交易信息、余额和脚本等内容，读者可访问 <https://github.com/mikispag/rusty-blockparser> 下载该解析器。该解析器采用 Rust 语言编写，提供了较为高级的区块链分析功能。这项工作的早期版本称作 BitIodine，但现已基本处于停顿状态。

为了解决比特币的隐私问题，研究人员已对此提出了各种建议。这些建议分为 3 类：混合协议、第三方混合协议和内在匿名性。下面对每种类型进行逐一讨论。

1. 混合协议

混合协议用于向比特币交易提供匿名特征。在该模型中，采用了混合服务提供者（中介或共享钱包）。作为存款，用户将货币发送到这一共享钱包中；随后，共享钱包可以向目的地发送其他货币（其他用户存储的等值货币）。另外，用户还可以通过中介接收其他人发送的货币。这样一来，输出和输入之间的链接就不复存在，交易图分析将无法显示发送者和接收者之间的真实关系。

混币原理（CoinJoin）是混合协议的一个例子。其中，两项交易在保持输入和输出不变的情况下，结合在一起形成一项交易。混币原理的核心思想是，构建由所有参与者签署协议的共享交易。这项技术提高了对所有交易参与者的隐私保护程度，如图 5-3 所示。

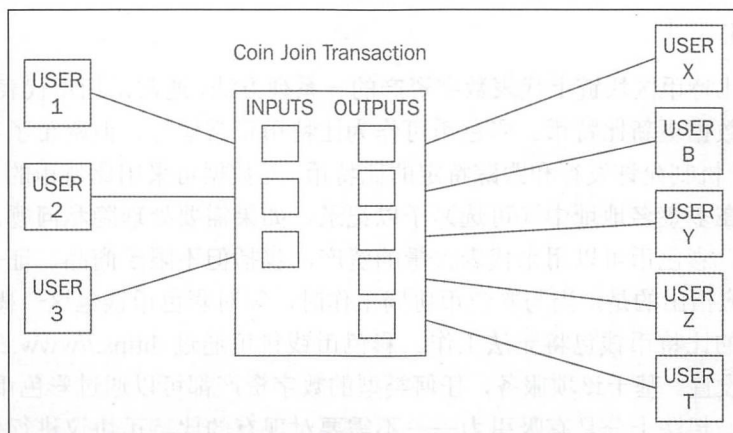


图 5.3 混币交易。其中，3 个用户将其交易加入至较大的单一混币交易中

2. 第三方混合协议

当前存在各种第三方混合服务，但如果服务具有中心化特征，则会受到跟踪发送者-接收者映射这一类威胁，此时混合服务知晓所有输入和输出内容。除此之外，完全中心化的矿工甚至会遭受被服务管理者偷取货币的风险。

包含不同复杂度的不同的服务，如匿名混币、Coinmux 以及达世币中的 Darksend，均建立于混合币交易这一基础之上。其中，匿名混币是传统混合服务的一种去中心化的替代方式，因为该方案不需要任何受信第三方的加入。

然而，混合币方案也存在一些弱点，其中最突出的是用户可能会遭受“拒绝服务”攻击：用户初始时承诺签署这些交易，但当前并未提供其签名，从而推迟或终止联合交易。

3. 内在匿名性

该类型包括内在支持隐私的货币，且内置在货币的设计过程中。其中，最受欢迎的是 Zcash，稍后将对此详细讨论。其他的例子还包括门罗币（Monero），并利用环签名提供匿名服务。

下面将讨论各种增强措施，进而对比特币协议进行扩展。

5.2.2 比特币上的扩展协议

为了进一步完善、扩展比特币协议，同时扩大其应用范围（而非仅是虚拟货币），比特币之上还实现了多种其他协议。

1. 彩色币

彩色是在比特币区块链上代表数字资产的一系列方法，通常是指用代表数字资产（智能财产）的元数据更新比特币。彩色币可作为比特币正常运行，但附带了某些代表资产的元数据。这一机制允许发行和跟踪特定的比特币。元数据可采用比特币的 `OP_RETURN` 操作码，或者在多签名地址中（可选）予以记录。如果需要处理隐私问题，也可以对元数据进行加密。彩色币可以用来代表大量的资产，包括但不限于商品、证书、股票、债券和投票。还应指出的是，当与彩色币协同工作时，须对彩色币钱包这一概念加以解释；相应地，普通的比特币钱包将无法工作。彩色币钱包可通过 <https://www.coinprism.com/> 服务实现在线设置。基于该项服务，任何类型的数字资产都可以通过彩色币创建和发行。

彩色币这一想法十分具有吸引力——不需要对现有的比特币协议进行任何修改，而且可以利用现有的安全比特币网络。除了数字资产这种传统表达方式之外，还可创建根据所定义的参数和条件运行的智能资产。这些参数包括时间验证、对可转移性的限制以及费用。这也体现了创建智能合同的可能性。

区块链上发行金融工具则是彩色币的一个主要应用，同时可确保较低的交易费用、有效且具有数学保证的所有权证明、无中介的快速转让性，并即时向投资者支付红利。

读者可访问 <http://coloredcoins.org/> 以获取丰富的彩色币 API。

2. 合约币

该项服务可以用于创建用作加密货币的定制令牌并还有多种用途，例如，在比特币区块链上发布数字资产。这是一个相当强大的平台，可运行于比特币区块链的核心上；同时开发了自己的客户端和其他组件以支持发布数字资产。

该架构由合约币的服务器、区块、钱包和 `armory_utxsvr` 组成。合约币的工作原理与彩色币一样，将数据置入普通的比特币交易中，但提供了一个更丰富的库，以及一组强大的工具，以支持数字资产的处理。这种置入行为也称作嵌入式共识，此时合约币交易嵌入在比特币交易中。相应地，在比特币中嵌入数据的方法是使用 `OP_RETURN` 操作码完成的。

合约币生成和使用的货币称作 XCP，并被智能合约用作履行合同的费用。在本书编写时，XCP 的价格是 2.78 美元。另外，XCP 采用之前所讨论的燃烧证明方法予以创建。合约币允许使用 Solidity 语言在 Ethereum 上开发智能合约，并可与比特币区块链进行交互。为了实现这一点，在以太坊和比特币之间，BTC Relay 可视为二者间互操作的一种手段。通过 BTC Relay，以太坊合约可与比特币区块链和交易进行通信，运行 BTC Relay 的节点可获取比特币区块头，并将其转至验证 PoW 的以太坊网络上的智能合约。该过程

验证了在比特币网络上产生的交易。读者可访问 <http://btcrelay.org/> 以获取更多内容。

从技术上讲，这基本上可视作存储和验证比特币区块头的以太坊合约，类似于基于 Bloom 过滤器的比特币支付验证客户端（轻量级）。SPV 客户曾在第 4 章有所讨论。当前思想可采用图 5.4 来表示。

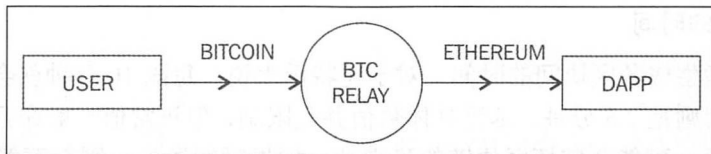


图 5.4 BTC Relay 概念



注意：

读者可访问 <http://counterparty.io/> 以了解合约币。

5.2.3 替代币的开发

从编码角度来看，启动替代币项目并不复杂，利用比特币或另一种货币的源代码即可，但实际过程远不止于此。当启动一个新的货币项目时，有几件事需要考虑，以保证该货币的有效性和寿命。一般情况下，代码库是采用 C++ 语言编写的，就像比特币一样，但几乎任何语言都可以用于开发货币项目，例如 Golang 或 Rust 语言。

编写代码或使用现有货币的源代码并不是这里所要讨论的重点，如何启动一种新货币往往是一个具有挑战性的问题，进而吸引新的投资者和用户。一般来说，启动新的货币项目一般需要采取多个步骤，稍后将对此予以解释。

从技术角度来看，当使用另一种货币（例如比特币）的代码时，须对多种参数进行调整，以有效地构造一种新型货币。相关参数包括但不限于以下内容。

1. 共识算法

共识算法的选择种类包括：用于比特币中的工作量证明（PoW），以及 Peercoin 中的权益证明（PoS）。

2. 哈希算法

哈希算法包括 SHA256、Scrypt、X11、X13、X15，以及针对共识算法应用中的其他相关哈希算法。

3. 难度调整算法

难度重定向机制包含多种选择,例如 KGW、DGW、Nite 推出的 Gravity Wave,以及 DigiShield。此外,所有这些算法都可以根据要求进行调整,以产生不同的结果。因此,可能会存在多个版本。

4. 区块间的时间

该术语是指生成各区块间的时间。对于比特币来说,每隔 10 分钟就会产生比特币,而莱特币的时间则是 2.5 分钟。尽管具体数值并无限制,但正常值一般在几分钟之内。如果生成时间太短,可能会破坏区块链的稳定性;如果时间过长,则有可能无法吸引更多用户。

5. 区块奖励

区块奖励主要面向求解了采矿谜题的矿工,同时产生一个包含该奖励的 Coinbase 交易。开始时,区块奖励一般是 50 个比特币;当前,许多替代币将这一参数值设置为一个很高的数字。例如在 Dogecoin 中,对应值为 10000。

6. 奖励减半速率

减半速率也是较为重要的一个因素。相应地,比特币的奖励数量每隔 4 年减半,当前值为 12.5。根据实际需求条件,这一变化中的年限值可设置为任意周期,或者不对其进行设置。

7. 区块大小和交易规模

在替代币中,该参数随着具体情况变化。

8. 利率

该属性仅适用于 PoS 系统,其中,货币持有者可通过网络定义的速率赚取利息;作为回报,网络上持有的货币量将作为一个 PoS 保护网络。

9. 货币期限

这个参数定义了货币要花费多长时间方可被认为是具有一定价值。

10. 全部货币供给量

这一数字设置了所生成的货币的总极限。例如,在比特币中,这一限制数字是 2100 万;而在 Dogecoin 中,则对此不做任何要求。另外,这一限制值是通过之前讨论的区块奖励和减半计划来确定的。

虚拟货币构造包含两种选择:利用现有的加密货币源代码,或者从头开始编写新代

码。其中，后者较少使用，而前者则相对易于实现，并在过去的几年中创建了大量的虚拟货币。从根本上讲，这一理念首先借鉴了加密货币的源代码，然后在源代码中的重要位置处进行适当的调整，从而创建一种新型货币。

下一节将介绍一些替代币项目。需要说明的是，本章不可能涵盖所有替代币，但会根据币龄、市场上限和创新行为有选择性地加以介绍，包括每一种货币的理论基础、交易和挖掘等不同角度。

5.3 域名币

域名币是比特币源代码的第一个分支，背后的关键思想并不是构造一种替代币，而是提供改进的去中心化机制、防审查机制、隐私、安全以及更快的去中心化命名机制。去中心化命名服务旨在向某些固有的局限性提供响应，例如传统域名系统（DNS）协议中的迟缓和集中控制等现象。另外，域名币也是 Zooko 三角关系的第一个解决方案，之前曾对此有所讨论。

域名币主要用于提供一项服务以注册一个键/值对，其主要示例包括：提供去中心化的传输层安全（TLS）证书验证机制，并通过基于区块的分布式和去中心化的共识所驱动。

域名币采用了与比特币相同的技术，但拥有自己的区块链和钱包软件。读者可访问 <https://github.com/namecoin/namecoin-core> 以了解域名币的核心源代码。

综上所述，域名币提供了以下 3 项服务：

- ❑ 安全存储和名称（键）传输。
- ❑ 通过（最多）添加 520 个数据字节，实现键-值绑定。
- ❑ 生成数字货币（域名币）。

域名币首次引入了合并挖掘机制，某个矿工可同时拥有多个链。这一想法简单、有效：矿工创建一个域名币区块，并生成该块的哈希值。随后，可将哈希值添加到比特币区块中。同时，矿工以域名币同等难度（或大于该难度）处理当前区块，进而证明已经为处理域名币区块提供了足够的工作量。

更准确地说，币基（Coinbase）交易包括源自域名币（或其他任何替代币）的交易哈希值。相应地，挖掘任务则负责处理比特币区块，其币基 scripSig 包含一个指向域名币（或其他替代币）区块的指针。

更准确地讲，币基交易包含了源自域名币（或其他替代币）的交易哈希值。同时，挖掘任务将处理比特币区块，其币基 scripSig 定义了一个指向域名币（或其他任何替代币）

的哈希指针，如图 5.5 所示。如果矿工尝试在比特币区块链难度级别上求解哈希值，将构造比特币区块且该区块形成比特币网络中的比分内容。此处，域名币将被比特币区块链忽略。另外一方面，以域名币区块链难度级别处理某一区块，新区块将在域名币区块链中构建。该方案的核心优势在于，矿工占用的全部计算能力将有助于向域名币和比特币提供安全保障。

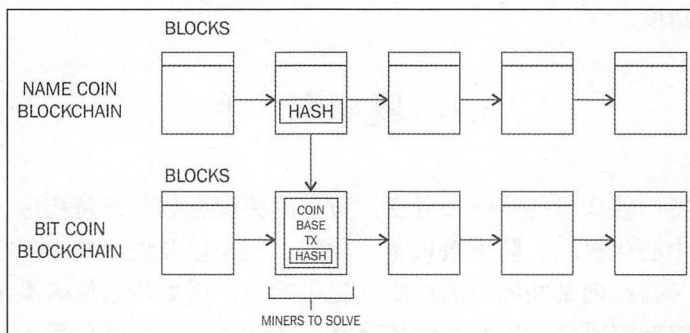


图 5.5 合并挖掘

1. 域名币交易

域名币当前市值为 2736537 英镑（数据源自 <https://coinmarketcap.com/>），并可以在不同交易平台进行买卖，例如 <https://cryptonit.net/>。除此之外，通过简单的在线搜索还可以找到其他交易方式。

2. 获取域名币

即使域名币可独立进行挖掘，但通常作为比特币中的一部分，并采用前述合并挖掘技术进行挖掘。通过这种方式，域名币可作为比特币挖掘的副产品。另外，独立挖掘将难以获取利润，如图 5.6 所示。相反，建议采用合并挖掘技术并使用矿池；甚至还可通过加密货币交换来购买域名币。



图 5.6 域名币的难度级别，数据来自 <https://bitinfocharts.com/comparison/difficulty-nmc.html>

另外, 各种矿池也提供了合并挖掘过程中的多种选项, 例如 <https://slushpool.com>。其中, 矿工首先挖掘比特币, 同时也会获取域名币。

另一种快速获取域名币的方法是将现有货币与域名币交换, 例如比特币或其他加密货币, 即可用来与域名币交换。对此, 一些在线服务中提供了此项服务, 如 <https://shapeshift.io/>。该服务提供了简单友好的界面, 并可将一种加密货币转换为另一种加密货币。

图 5.7 显示了 BTC 与域名币之间的兑换方式。

Instant Rate 1 BTC = 3114.84374999 NMC

Deposit Min	Deposit Max	Liquidity
0.00000300 BTC	0.14532464 BTC	00000

Bitcoin → Namecoin

NFgrujLsjwRGWtRFJ25RNFUqUcjs9Fsgb

14Koadj8xLpAeKDFke8qVWX5ETeU81amxH

☒ I agree to Terms

Miner Fee: NMC

Start Transaction

图 5.7 BTC 与域名币之间的兑换

单击 Start Transaction 后, 交易即启动, 并指示用户将比特币发送至特定的比特币地址。当用户提交了所需额度后, 转换过程如图 5.8 所示。

处理过程结束后, 交易结果显示于域名币钱包中, 如图 5.9 所示。

交易确认过程可能会占用一些时间。在此之前, 将无法使用域名币管理名称。一旦域名币在钱包中可用, Manage Names 选项即可用于生成域名币记录。

3. 生成域名币记录

域名币记录采用了键/值对形式。其中, 名称是形如 d/examplename 的小写字字符串; 而值则是一个区分大小写的 UTF-8 编码 JSON 对象, 最大为 520 字节。同时, 对应名字还应兼容于 RFC1035 (参见 <https://tools.ietf.org/html/rfc1035>)。

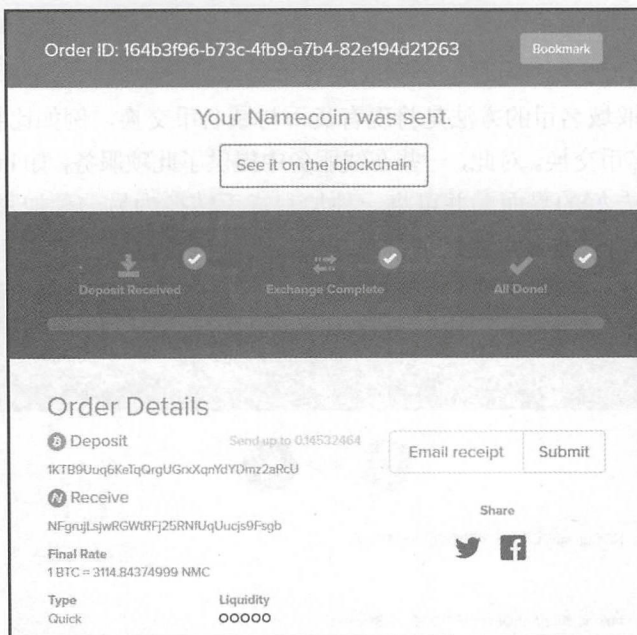


图 5.8 转换处理

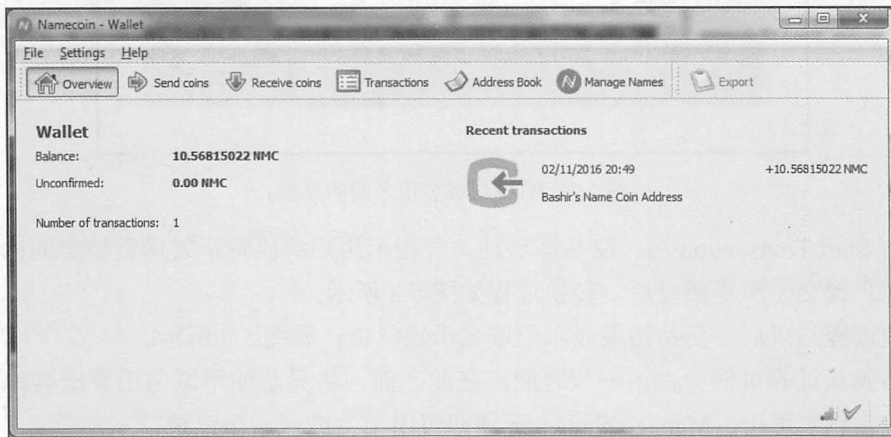


图 5.9 交易结束

通用的域名币名称可表示为任意的二进制字符串（长达 255 字节），以及 1024 位的关联识别信息。域名币链上的一个记录只能在大约 200 天或 36000 个块之后才有效，随后即需要进行更新。除此之外，域名币还引入了 .bit 顶级域名，可通过域名币进行注册，并采用专门的域名币解析器浏览。在图 5.10 中，域名币钱包软件可用于注册 .bit 域名。

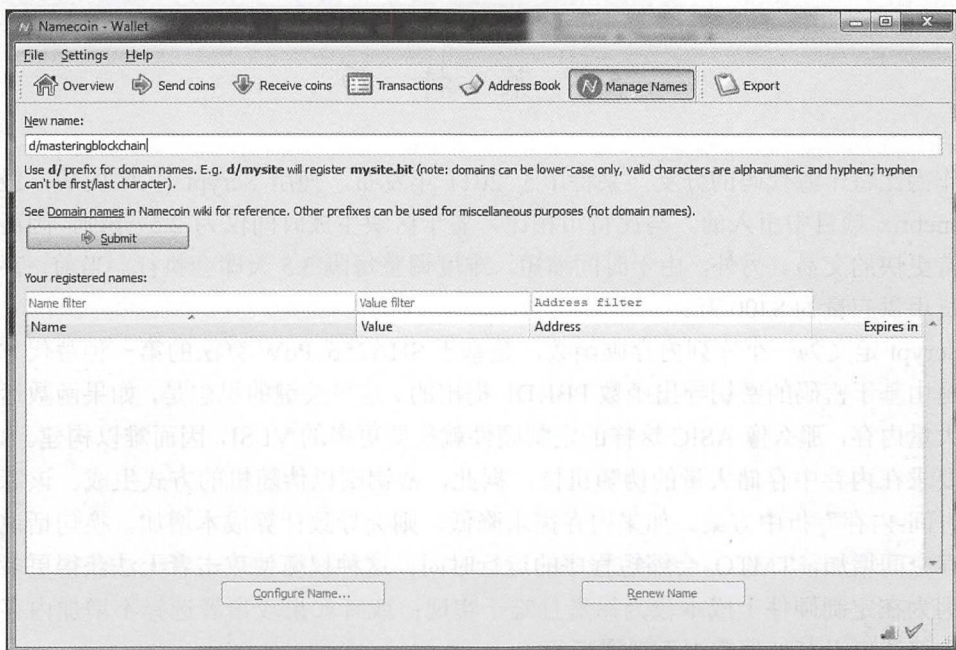


图 5.10 域名注册

当输入名称并单击 Submit 按钮后，将请求 DNS、IP 或 Identity 配置信息。

在图 5.11 中，masteringblockchain 在域名币区块链上注册为 masteringblockchain.bit。

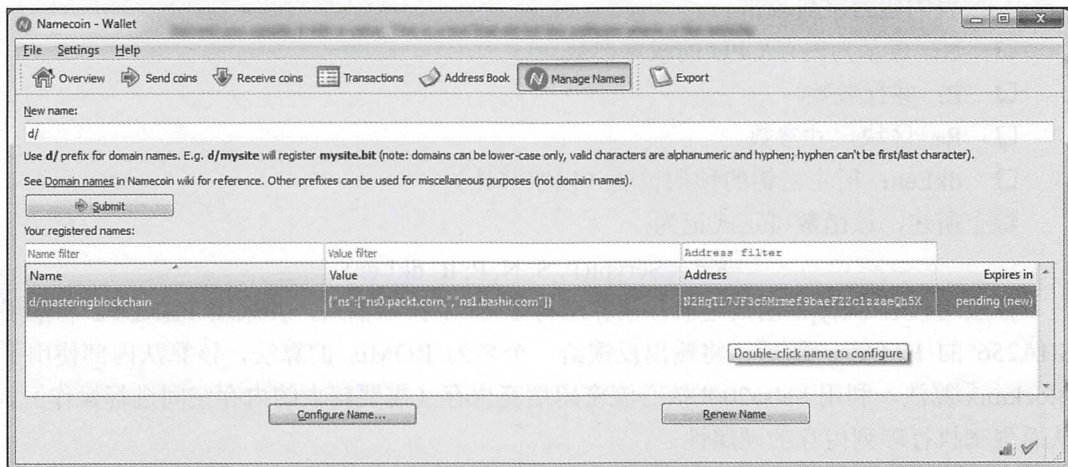


图 5.11 域名币区块链上的注册

5.4 莱 特 币

作为比特币源代码的分支，莱特币于 2011 年发布，使用 Scrypt 作为 PoW，最初是在 Tenebrix 项目中引入的。与比特币相比，鉴于区块生成时间仅为 2.5 分钟，因而莱特币支持更快的交易。另外，由于时间缩短，难度调整每隔 3.5 天即会执行。当前，莱特币的总货币供应量为 8400 万。

Scrypt 定义为一个序列内存硬函数，是基于 SHA256 PoW 算法的第一种替代方法，最初是由基于密码的密钥导出函数 PBKDF 提出的。这里关键的思想是，如果函数运行时占用大量内存，那么像 ASIC 这样的定制硬件就需要更多的 VLSI，因而难以构建。Scrypt 算法要求在内存中存储大量的伪随机位，据此，密钥则以伪随机的方式生成。该算法基于“时间-内存”折中方案。如果内存需求降低，则会导致计算成本增加。换句话说，如果内存空间增加，TMTO 会缩短程序的运行时间。这种权衡使攻击者无法获得更多的内存，因为在定制硬件上成本较为昂贵且难于实现；或者如果攻击者选择不增加内存，则会导致算法由于高处理需求而缓慢运行。

Scrypt 采用下列参数生成衍生密钥 (Kd)。

- ❑ **Passphrase**: 执行哈希运算的字符串。
- ❑ **Salt**: 向 Scrypt 函数提供的随机字符串（通常为哈希函数），进而防御使用彩虹表的蛮力字典攻击。
- ❑ **N**: 表示为内存/CPU 的成本参数。
- ❑ **P**: 并行参数。
- ❑ **R**: 区块尺寸参数。
- ❑ **dkLen**: 衍生密钥的预期长度（以字节计）。

综上所述，该函数可正式记为：

$$Kd = \text{Scrypt}(P, S, N, P, R, dkLen)$$

在应用核心 Scrypt 函数之前，该算法将 P 和 S 作为输入，并采用 PBKDF2 和基于 SHA256 的 HMAC。随后，将输出反馈给一个名为 ROMix 的算法，该算法内部使用了 Blockmix 算法，利用 Salsa20/8 核心流密码填充内存（需要较大的内存空间进行操作），从而强制执行序列内存的硬属性。

源自算法当前步骤的输出结果将再次反馈至 PBKDF2 函数中，进而输出衍生密钥。对应处理过程如图 5.12 所示。

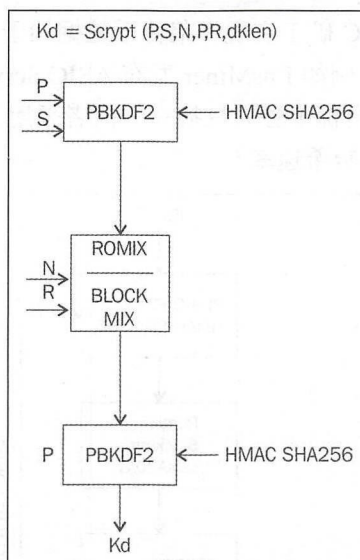


图 5.12 Scrypt 算法

Scrypt 用于莱特币的挖掘过程，其中，特定的参数表示为 $N=1024$ ， $R=1$ ， $P=1$ ， S 则表示为随机的 80 个字节，并生成 256 位的输出结果。

由此看来，根据上述各项参数，ASIC 的构建并不十分困难。在基于莱特币挖掘的 ASIC 中，可以将数据和 nonce 作为输入，并通过 HMAC-SHA256 使用 PBKDF2 算法。然后，将生成的位流输入至 SALSA20/8 函数，该函数生成一个哈希值后再次输入到 PBKDF2 和 HMAC-256 函数中，以生成一个 256 位的哈希输出结果。与比特币 PoW 一样，在 Scrypt 中，如果输出哈希值小于目标哈希值（在开始阶段已经作为输入传递，存储在内存中，并在每次迭代中进行检查），则函数终止；否则，nonce 将增加。该过程重复执行，直到发现一个哈希值低于难度目标。全部流程如图 5.13 所示。

1. 莱特币交易

与其他货币一样，可在各种在线平台上实现莱特币交易。目前，莱特币的市值为 161239005 英镑，其价格为 3.25 英镑/LTC。

2. 莱特币挖掘过程

莱特币的挖掘过程可采用独立方式或在矿池中进行。当前，基于 Scrypt 的 ASIC 常用于莱特币挖掘。

在 CPU 上采矿莱特币不再像其他数字货币一样有利可图。莱特币的开采过程可通过

在线云服务挖掘提供商和 ASIC 矿工实现。莱特币挖掘始于 CPU，历经了 GPU 开采，最终通过特定的矿工生成货币，例如 EhsMiner 发布 ASIC Scrypt Miner Wolf。通常情况下，即使采用 ASIC，矿池挖掘也会优于独立挖掘——前者采用了比例回报方案。对于 Scrypt 而言，矿工能够产生 2Gh/s 的哈希速率。

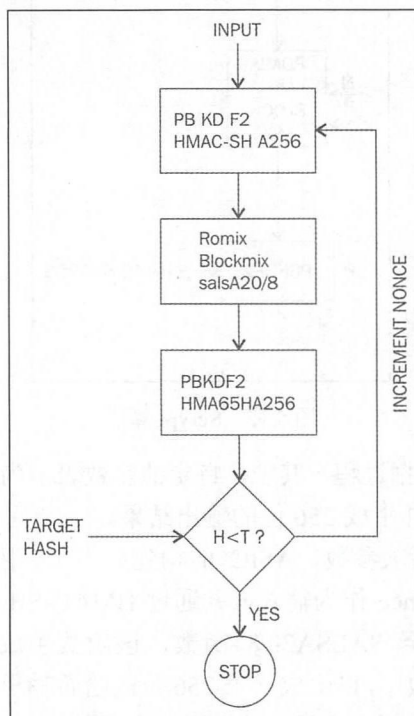


图 5.13 简化的 Scrypt ASIC 设计流程图

3. 软件源代码和钱包

读者可访问 <https://github.com/litecoin-project/litecoin> 下载莱特币源代码；而莱特币钱包则可在 <https://litecoin.org/> 处下载，其使用方式类似于比特币核心客户端软件。

5.5 素数币

素数币是市场上第一个引入 PoW 的数字货币，而非比特币中基于 SHA256 的 PoW。素数币采用质数搜索作为 PoW。另外，不是所有类型的质数都符合相关要求并被选为

PoW。相应地,存在3种类型的素数(即第一类 Cunningham 链,第二类 Cunningham 链,以及 bi-twin 链)可满足 PoW 算法需求条件,并在加密货币中予以使用。通过域名币区块链中的连续难度评估方案,难度可实现动态调整。基于素数的 PoW 高效验证同样十分重要——如果验证过程较慢,那么 PoW 将不再适用。因此,素数链可选择为 PoW,其原因在于,随着链长度增加,获取素数将变得越发困难,而验证过程仍保持应有的速度,从而确保用作有效的 PoW 算法。同样重要的是,一旦 PoW 在某个区块上被验证,将不能在另一个区块上重复使用。通过工作量证明证书,并利用子区块中父区块头对其进行哈希计算,该过程即可在素数币中完成。另外,将素数链链接至区块头哈希值可生成 PoW 证书。除此之外,还要求区块头的起源可被区块头哈希值整除。如果可整除,则商表示为 PoW 证书。PoW 算法可调难度的另一个特性是,在每个区块上都引入了难度调整,而不是比特币中每隔 2016 块方引入难度调整。与比特币相比,素数币则是一种更平滑的方法,在哈希值突然增加的情况下兼具重新调功能。此外,产生的货币总数是由社区驱动的,且素数币所能产生的货币数量没有具体限制。

5.5.1 素数币交易

素数币可以在主要的虚拟货币交易平台上进行交易。在本书编写时,素数币的市值为 828002 英镑,如图 5.14 所示。虽然这一数字并不是很大,但素数币涵盖了较新的理念,并且背后存在一个专门的社区作为支持,因而在未来一段时间内仍会继续占据一些市场份额。

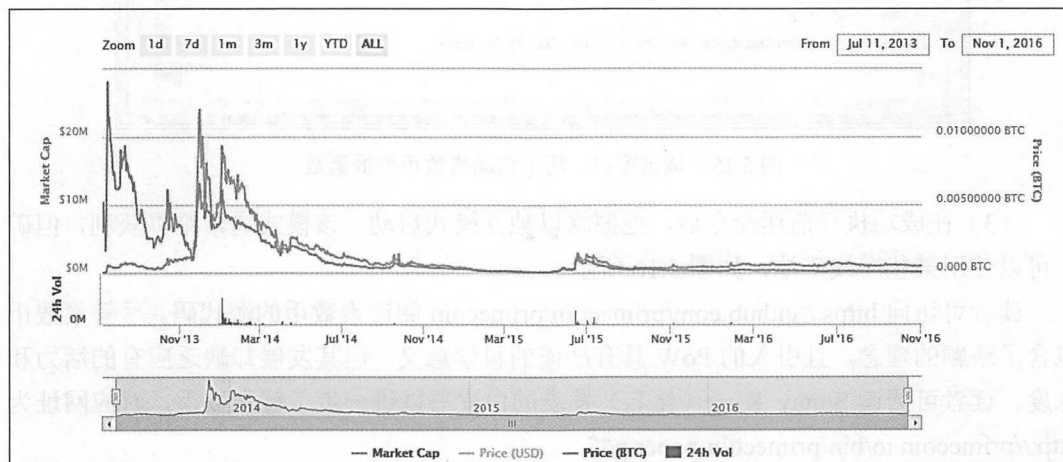


图 5.14 素数币的市值

5.5.2 挖掘规则

首先需要下载一个钱包软件。Primecoin 支持钱包内的原生挖掘，这一点与最初的比特币客户端一样，但也可以通过各种在线挖掘服务提供商在云中进行挖掘。

简而言之，挖掘规则包含如下步骤：

- (1) 下载素数币钱包。对此，读者可访问 <http://primecoin.io/index.php>。
- (2) 当钱包安装完毕并与网络同步后，在下一个步骤中即可启动挖掘操作。选择“帮助”→“调试窗口”命令，可以在素数币钱包中打开一个调试窗口。另外，在控制台窗口中输入 help 即可查看帮助内容，如图 5.15 所示。

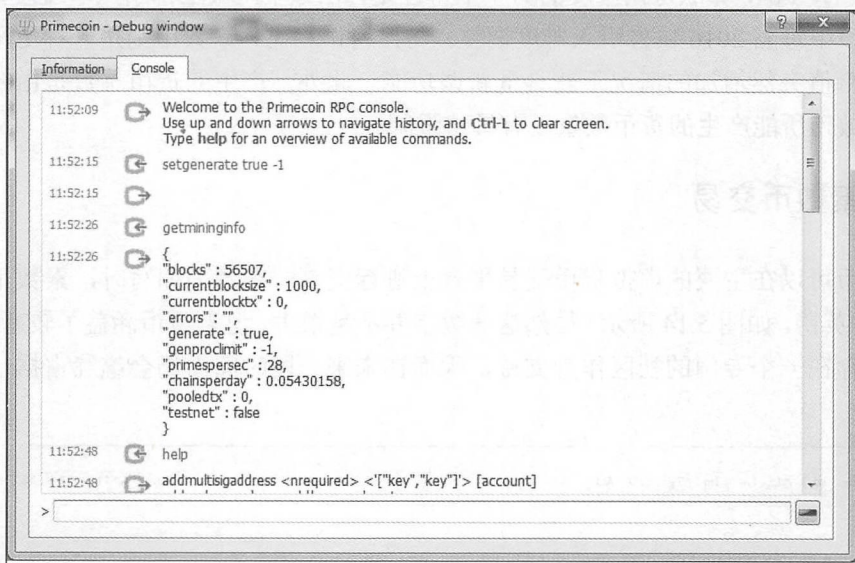


图 5.15 调试窗口，用于启动素数币挖掘函数

(3) 在成功执行前述命令后，挖掘将以独立模式启动。该模式通常难以获利，但矿工可以使用某些在线矿池，如图 5.16 所示。

读者可访问 <https://github.com/primecoin/primecoin> 阅读素数币的源代码。尽管素数币包含了新颖的理念，且引入的 PoW 具有严谨的科学意义，但其发展却缺乏应有的活力和深度。读者可阅读 Sunny King（化名）发表的白皮书以进一步了解素数币，对应网址为 <http://primecoin.io/bin/primecoin-paper.pdf>。

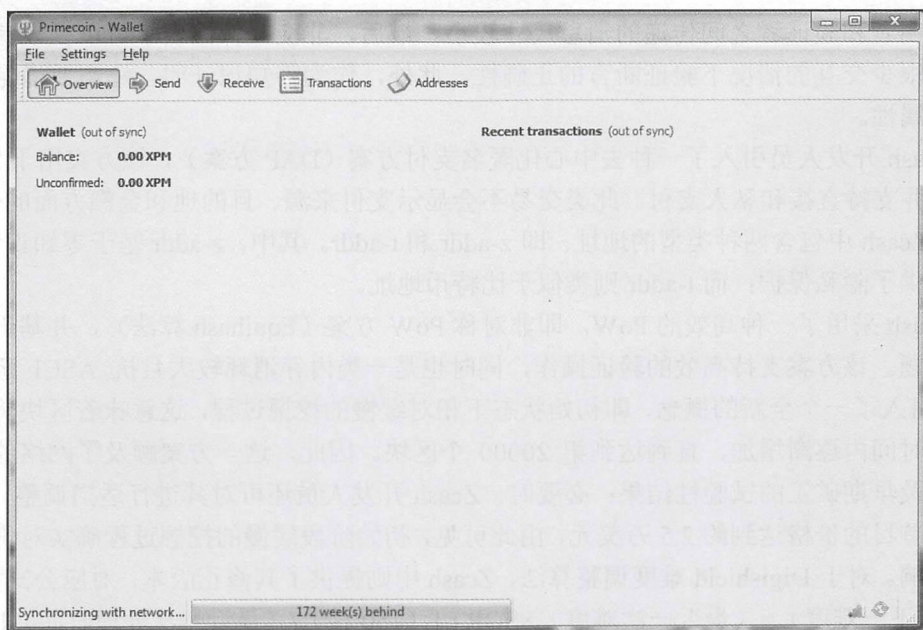


图 5.16 素数币钱包软件并与网络同步

5.6 Zcash

Zcash 于 2016 年 10 月 28 日推出，也是第一个使用特定类型的零知识证明的货币，即零知识简明非交互式知识论（zk-SNARK），并向用户提供完整的隐私性。这一类证明较为简短且易于验证。然而，设置初始的公有参数则是一个复杂的过程，其中包括两个密钥：证明密钥和验证密钥。该过程需要采样一些随机数以构造公有参数。这里的问题是，随机数（也称作有害数据）必须在参数生成后销毁，以防止伪造 Zcash。为此，Zcash 团队提出了一个多方计算协议，以协作方式从无关位置生成所需的公有参数，以确保不会产生有害数据。由于这一类公有参数由 Zcash 团队创建，这也意味着该过程的参与者可信，因而具有开放性特征，并在多方计算机制的指导下进行。该机制具备一种特征，即过程中的全部参与者须达成妥协，以形成最终的参数。结束后，全部参与者须物理销毁生成密钥的设备。这一行为可消除设备上密钥参与者的痕迹。

zk-SNARK 须具备完备性、可靠性、简洁性以及非交互特征。这里，完备性意味着，对于证明者而言应存在一个明确的策略，以满足断言为真的验证者；另外一方面，稳定性则表示，对于假命题为真这种情况，不存在证明者可“说服”验证者；而简洁性意味

着在证明者和验证者之间传递的消息尺寸较小。最后，非交互特性表示可以在没有任何交互或很少交互的情况下验证断言的正确性。此外，作为零知识证明，还需要满足其中的各项属性。

Zcash 开发人员引入了一种去中心化匿名支付方案（DAP 方案），该方案用于 Zcash 网络，并支持直接和私人支付。此类交易不会显示支付来源、目的地和金额方面的信息。另外，Zcash 中包含两种类型的地址，即 z-addr 和 t-addr。其中，z-addr 基于零知识证明，同时提供了隐私保护；而 t-addr 则类似于比特币地址。

Zcash 采用了一种高效的 PoW，即非对称 PoW 方案（Equihash 算法），并基于广义生日问题。该方案支持高效的验证操作，同时也是一类内存消耗较大且抗 ASCII 函数。Zcash 引入了一个全新的概念，即初始状态下相对缓慢的挖掘过程，这意味着区块奖励将在一段时间内逐渐增加，直到达到第 20000 个区块。因此，这一方案顾及了网络的初始规模以及早期矿工的试验性结果；必要时，Zcash 开发人员还可对其进行适当调整。ZEC 在上市首日的价格达到约 2.5 万美元，由此可见，初始阶段缓慢的挖掘过程确实对价格造成了影响。对于 Digishield 难度调整算法，Zcash 中则提供了其修正版本，对应公式如下：

$$(\text{下一难度}) = (\text{最近一次难度}) \times \text{SQRT}[(150 \text{ 秒}) / (\text{最近一次处理时间})]$$

在缓慢的初始启动阶段后，Zcash 中的各种属性如表 5.1 所示。

表 5.1 Zcash 中的属性

属 性	值
名称	Zcash
发布日期	2016 年 10 月 28 日
主要功能	货币
货币代码	ZEC
最大货币量	2100 万
区块时间	10 分钟
共识算法	工作量证明（Equihash）
难度调整算法	Digishield V3（修正版）
挖掘硬件	CPU 和 GPU
难度调整周期	1 个区块

5.6.1 Zcash 交易

Zcash 可以在主要的数字货币交易机构购买。在本书编写时，Zcash 的价格非常高，已飙升至每 Zcash 大约 10 个比特币，如图 5.17 所示。一些交易机构的订单高达 2500

BTC/ZEC。

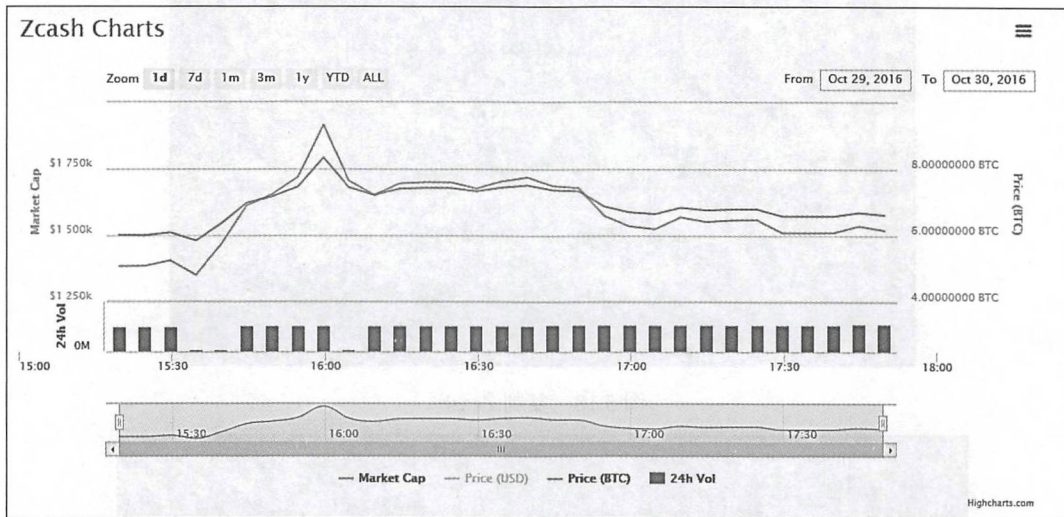


图 5.17 Zcash 市值

5.6.2 挖掘规则

Zcash 的挖掘过程包含多种方案，目前支持 CPU 和 GPU 挖掘。除此之外，各种商业云矿池也提供了 Zcash 挖掘合约。若执行基于 CPU 的独立挖掘方案，则须遵循下列步骤。

(1) 采用以下命令安装组件：

```
sudo apt-get install \
build-essential pkg-config libc6-dev m4 g++-multilib \
autoconf libtool ncurses-dev unzip git python \
zlib1g-dev wget bsdmainutils automake
```

待安装结束后，将显示一条消息，表明各组件已为最新版本。对于早期安装包，安装过程会持续进行，同时下载所需的数据包，直至安装结束。

(2) 通过 git 命令复制 Zcash，对应代码如下：

```
$ git clone https://github.com/zcash/zcash.git
```

该命令采用本地方式复制 Zcash git 存储库，输出结果如图 5.18 所示。

(3) 如图 5.19 所示，采用图中命令下载证明密钥和验证密钥。

```

drequinox@drequinox-OP7010:~$ git clone https://github.com/zcash/zcash.git
Cloning into 'zcash'...
remote: Counting objects: 56593, done.
remote: Total 56593 (delta 0), reused 0 (delta 0), pack-reused 56593
Receiving objects: 100% (56593/56593), 42.78 MiB | 2.11 MiB/s, done.
Resolving deltas: 100% (43020/43020), done.
Checking connectivity... done.
drequinox@drequinox-OP7010:~$ cd zcash/
drequinox@drequinox-OP7010:~/zcash$ git checkout v1.0.0
Note: checking out 'v1.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

HEAD is now at 1feaeafa... Update network magic's for 1.0.0

```

图 5.18 复制 Zcash

```

drequinox@drequinox-OP7010:~/zcash$ ./zcutil/fetch-params.sh
Zcash - fetch-params.sh

This script will fetch the Zcash zkSNARK parameters and verify their
integrity with sha256sum.

The parameters are currently just under 911MB in-size, so plan accordingly
for your bandwidth constraints. If the files are already present and
have the correct sha256sum, no networking is used.

Creating params directory. For details about this directory, see:
/home/drequinox/.zcash-params/README

Retrieving: https://z.cash/downloads/sprout-proving.key
--2016-10-28 21:46:21-- https://z.cash/downloads/sprout-proving.key
Resolving z.cash (z.cash)... 104.236.171.172
Connecting to z.cash (z.cash)|104.236.171.172|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://s3.amazonaws.com/zcashfinalmpc/sprout-proving.key [following]
--2016-10-28 21:46:22-- https://s3.amazonaws.com/zcashfinalmpc/sprout-proving.key
Resolving s3.amazonaws.com (s3.amazonaws.com)... 54.231.40.114
Connecting to s3.amazonaws.com (s3.amazonaws.com)|54.231.40.114|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 910173851 (868M) [application/octet-stream]
Saving to: '/home/drequinox/.zcash-params/sprout-proving.key.dl'

0K ..... 3% 2.71M 5m8s
32768K ..... 7% 3.58M 4m20s
65536K ..... 11% 2.53M 4m28s
98304K ..... 14% 1.75M 4m59s
131072K .....

```

图 5.19 下载证明密钥和验证密钥

(4) 随后将下载约 911MB 的密钥内容至~/zcash-params/目录中。该目录中包含了与证明密钥和验证密钥相关的文件，如下所示：

```

drequinox@drequinox-OP7010:~/zcash-params$ pwd
/home/drequinox/.zcash-params
drequinox@drequinox-OP7010:~/zcash-params$ ls -ltr

```



```
-rw-rw-r-- 1 drequinox drequinox 1449 Oct 24 16:46 sproutverifying.
key
-rw-rw-r-- 1 drequinox drequinox 910173851 Oct 24 16:46 sproutproving.
key
```

(5) 待上述命令执行完毕后, 可通过下列命令设置源代码:

```
./zcutil/build.sh -j$(nproc)
```

该命令将产生较长的输出结果。如果一切顺利, 最终将生成一个 Zcash 二进制文件。需要注意的是, 上述命令以 `nproc` 作为参数, 可获得系统中核数量或处理器的数量并予以显示。否则, 可将 `nproc` 替换为系统中的处理器数量。

构建完成后, 下一步是配置 Zcash。对此, 可在 `~/zcash/` 目录中生成包含 `zcash.conf` 名称的配置文件。

下列内容显示了相应的示例配置文件。

```
addnode=mainnet.z.cash
rpcuser=drequinox
rpcpassword=xxxxxxoJNo4o5c+F6E+J4P2C1D5iz1zIKPZJhTzdW5A=
gen=1
genproclimit=8
equihashsolver=tromp
```

上述配置内容支持各种特性。其中, 第一行代码添加了主干网节点, 并启用了主干网连接。其中, `rpcuser` 和 `rpcpassword` 表示 RPC 接口的用户名和密码; `gen = 1` 用于启动挖掘操作; `genproclimit` 表示可用于采矿的处理器数量。最后一行代码可以实现更快的挖掘解决方案, 对于标准的 CPU 挖掘, 则无此要求。

Zcash 可通过下列命令启动:

```
./zcashd --daemon
```

Zcash 启动后, 可通过 `Zcash-cli` 命令行界面与 RPC 界面进行交互, 且与比特币命令行界面基本类似。当运行 Zcash 守护进程时, 即可通过各种命令查询 Zcash 的不同属性。另外, 可使用 CLI 或通过区块链浏览器查看本地交易。读者可访问 <https://explorer.zcha.in/> 并下载 Zcash 区块链 Explorer 浏览器。

Z 地址可通过下列命令生成:

```
$:~/zcash/src$ ./zcash-cli z_getnewaddress
zcPDBKuwwHJ4gqT5Q59zAMXDHhFoihyTC1aLE5Kz4GwgUXfCBWG6SDr45SFLUsZhpcdvHt
7nFmC3iQcn37rKBcVRa93DYrA
```

执行包含 `getinfo` 参数的 `Zcash-cli` 命令将生成如图 5.20 所示的输出结果,其中包含了区块、难度以及余额等有用信息。

```
drequinox@drequinox-OP7010:~/zcash/src$ ./zcash-cli getinfo
{
  "version" : 1000050,
  "protocolversion" : 170002,
  "walletversion" : 60000,
  "balance" : 0.00000000,
  "blocks" : 601,
  "timeoffset" : 0,
  "connections" : 8,
  "proxy" : "",
  "difficulty" : 13748.56014152,
  "testnet" : false,
  "keypoololdest" : 1477688856,
  "keypoolsize" : 101,
  "paytxfee" : 0.00000000,
  "relayfee" : 0.00005000,
  "errors" : "WARNING: abnormally high number of blocks generated, 190 blocks received in the last 4 hours (96 expected)"
}
```

图 5.20 `getinfo` 参数的输出结果

下列命令可用于输出 T 地址:

```
drequinox@drequinox-OP7010:~/zcash/src$ ./zcash-cli getnewaddress
t1XRCGMAw36yPVCcxDUrxv2csAAuGdS8Nny
```

5.6.3 GPU 挖掘

除了 CPU 挖掘机制之外,还可采用 GPU 挖掘。对此,官方尚未发布 GPU 矿工,但开源工作者已对此进行了充分的准备。与此同时,Zcash Company 也鼓励开发人员构造并提交 CPU 和 GPU 矿工。但在本书编写时,事态尚无进展。关于 GPU 挖掘的更多信息,读者可访问 <https://zcashminers.org/>。

除此之外,还可使用在线云计算供应商提供的云挖掘合约。其中,云挖掘服务供应商代表客户进行挖掘。除了云挖掘合约之外,矿工还可使用自己的设备并通过矿池进行挖掘(基于层协议或其他协议),例如 Zcash 矿池。通过 Zcash 矿池,矿工还可出售其哈希数据。

通过下列各步骤可在 Ubuntu Linux 发行版上下载并编译 `nheqminer`。

```
sudo apt-get install cmake build-essential libboost-all-dev
git clone https://github.com/nicehash/nheqminer.git
cd dheqminer/nheqminer
mkdir build
cd build
cmake ..
make
```



```
./nhequminer -l eu -u <btc address> -t <number of threads>
```

nheqminer 可接收多个参数，例如位置 (-l)、用户名 (-u)，以及挖掘过程中的线程数量 (-t)。

[illegible]

图 5.21 利用 BTC 地址接收支付

[illegible]

图 5.22 利用 Zcash T 地址接收支付

读者可以在网络上搜索更多关于 Zcash 的信息。目前，Zcash 尚处于不稳定状态，而且变化非常迅速。但值得肯定的是，Zcash 的零知识证明可视为一项重大创新，并为内在隐私应用的发展铺平了道路，如银行、医药或法律等行业。

5.7 本章小结

本章对加密货币进行了整体介绍，同时也涉及了某些替代币，特别是 Zcash 和域名币。当前，加密货币是一个非常活跃的研究领域，在可伸缩性、隐私和安全方面尤其如此。为了消除加密货币中的中心化问题，人们投入了大量的精力并制定各种难度重定位算法。另外，在隐私和可扩展性方面，相关研究也从未停止。相信读者已对各种替代币的概念及其思想有所了解。除此之外，本章还讨论了一些实际操作内容，例如货币项目的挖掘和启动，并为读者今后的学习打下坚实的基础。总而言之，替代币是一个令人着迷的研究领域，在去中心化方面包含了多种可能性。

第6章 智能合约

本章将讨论智能合约，这并非是一个全新概念，但是随着区块链的出现，人们再次将目光转移到这一领域。同时，智能合约也是区块链研究中的一个活跃领域。鉴于智能合约针对金融服务行业产生成本节约效益，即降低交易成本并简化复杂合约，因而金融和学术机构均对此予以关注寄予厚望。

6.1 发展历史

早在 20 世纪 90 年代末，Nick Szabo 即提出了智能合约这一概念。待其潜力和优势充分发挥时，20 年已匆匆而过。Szabo 对智能合约提出了如下描述：“智能合约是一种计算机化的交易协议，可以执行合同的条款。一般的目标是满足常见的契约条件（例如支付条款、扣押权、机密性，甚至是强制执行），将恶意攻击和意外情况降低至最小，并最小化对受信中介的需求。相关的经济目标包括降低欺诈损失、仲裁和执行成本，以及其他交易成本。”

2009 年，智能合约这一概念仅在比特币中以一种有限的方式实现。其中，比特币交易可以用来在用户之间转移价值。在 P2P 网络中，用户之间无须相互信任，也不需要受信的中介。

6.2 定义

关于智能合约的标准定义，目前尚未达成共识。智能合约的定义具有重要意义，下面尝试提供一种智能合约的广义定义。



注意：

智能合约是一种安全的、处于运行状态的计算机程序，体现了一种自动执行以及强制执行的协议。

通过进一步分析可知，智能合约实际上是采用某种语言编写的、计算机或目标机可理解的计算机程序。此外，智能合约还包含了以业务逻辑形式出现的各方之间的协议。

另一个核心理念是，当某些条件满足时，智能契约将自动执行。另外，强制性意味着所有的合约条款都是按照规定和预期执行的，对于攻击者而言同样如此。这里，强制执行是一个相对广泛的术语，包括法律形式的传统强制执行，以及执行某些措施和控制行为，在无须中介的情况下使执行合同条款成为可能。需要注意的是，真正的智能合约不应依赖于传统的执法手段。相反，应该按照“代码即是法律”这一原则运作。这意味着，没有必要让仲裁人或第三方来控制或影响智能合约的执行。智能合约具有“自我执行”这一类特点，而不是法律意义上的强制执行。该方式可能被视为自由意志主义者的梦想，但这一结论完全可行，并且符合智能合约的真正精神。

除此之外，智能合约还应具备安全性以及“不可阻挡”性，也就是说，计算机程序必须以这样一种方式设计：在合理的时间范围内具备容错能力并保持执行状态。即使外部因素处于不利状态下，相关程序也应能够执行和维护一个健康的内部状态。例如，假设某个正常的计算机程序采用某种逻辑编码，并根据内部编码指令执行，如果运行环境或依赖的外部因素偏离正常或预期状态，该程序将处于不确定状态或简单地终止。更为重要的是，智能合约不会受到这一类问题任何影响。

安全性和不可阻挡性可视为一种需求条件或期望特性，但在开始阶段即将安全性和不可阻挡这一类属性纳入智能合约定义中，那么从长远角度来看，这将带来更大的收益。因此，研究人员从前期即可关注这方面内容，并为后续研究打下坚实的基础。同时，一些研究人员也提出，智能合约不必自动执行；相反，智能合约应具备所谓的自控性，因为在某些场合下仍需要人工输入。虽然在某些情况下，手工输入和控制是可取的，但这并非是绝对必要条件；而且，对于真正的智能合约，在本人看来，必须是完全自动化的。某些需要通过手动方式提供的输入，也应该通过 Oracle 等实现自动化。Oracle 将在后续内容中详细讨论。

智能合约一般通过状态机模型管理其内部状态，对于智能合约程序设计而言，该模型可实现高效的框架；相应地，合约状态根据一些预定义的标准和条件进一步推进。

在法律界，代码是否可作为合同尚存在争论，这与传统的法律条文是完全不同的。尽管智能合约确实代表并执行所有的合同条款，但是法院并不理解代码。因此，问题也随之而来：智能合约如何具有法律约束力？如何使智能合约在法庭上容易被接受和理解？如何在代码中实现争端的解决方案，且是否具有可行性？此外，在智能合约可以像传统的法律文件使用之前，监管和法规则是另一个需要解决的问题。上述问题包含了各种可能性，例如，让智能合约代码不仅可以被机器阅读，还可以被人类阅读。如果人类和机器都能理解所编写的代码，那么在法律环境中，智能合约可能更容易被接受，而不是仅供程序员阅读。此类特性在研究领域也日趋成熟，研究人员也耗费了大量的精力以

处理有关合约语义、含义以及解释性问题。

从本质上讲, 智能合约应具备确定性。据此, 网络上的任何节点运行同一个智能合约, 并获得相同的结果。如果最终结果在节点之间略有不同, 那么将无法达成共识。因此, 区块链上分布式共识的整个范例可能会失效。此外, 合约语言本身也是确定的, 从而确保智能合约的完整性和稳定性。根据个人经验, 确定性的含义是指语言中不存在非确定性函数, 进而导致在不同的节点上产生不同的结果。例如, 当采用不同编程语言中的函数进行计算时, 各种浮点操作可以在不同的运行环境中产生不同的结果。另一个例子是 JavaScript 中的一些数学函数, 这一类函数可以在不同的浏览器上产生不同的结果, 进而产生各种各样的 bug。这一类问题在智能合约中是非常致命的——如果节点之间的结果不一致, 那么就无法达成共识。这里, 确定性确保智能合约总是为特定的输入产生相同的输出。换言之, 编译后的程序将生成一个可靠、准确的业务逻辑, 同时完全符合在高级代码中编写的各项需求。

综上所述, 智能合约包含以下 4 项属性:

- ☐ 自动执行。
- ☐ 强制执行。
- ☐ 明确的语义。
- ☐ 安全性和不可阻止性。

其中, 前两个属性仅为最低要求, 而后两个属性在某些场合下并非必需, 因而可适当放宽条件。例如, 衍生合约可能不需要语义限制以及不可阻止特征, 但至少应在基本层面上实现自动执行和强制执行。另一方面, 合约需要具备语义上的明确性和完整性。因此, 智能合约在其实现过程中, 其应语言须被计算机和人类所理解。Ian Grigg 在其“李嘉图合约”中针对这一问题提供了相关解释, 稍后将对此加以讨论。

6.3 李嘉图合约

20 世纪 90 年代末, 李嘉图合约最初在 Ian Grigg 发表的一篇名为 *Financial Cryptography in 7 Layers* 的论文中提出, 并用于债券交易和支付系统中。其核心理念可描述为: 编写一份法律和计算机软件均可理解和接受的文档。对此, 李嘉图合约解决了在互联网上价值发行所面临的挑战, 并在一份文档中验证发行者, 同时获取合同的所有术语和条款, 以便使其成为具有法律约束力的合约。

根据 Ian Grigg 提出的原始定义 (参见 http://iang.org/papers/ricardian_contract.html), 李嘉图合约定义为一份文档, 并涵盖了以下属性:

- ❑ 发行者向持有者提供的一份合约。
- ❑ 持有者拥有价值权利，并由发行者所管理。
- ❑ 具有较好的可读性（类似于常见的纸质合同）。
- ❑ 具有程序可读性（类似于数据库）。
- ❑ 采用数字签名。
- ❑ 携带密钥和服务器信息。
- ❑ 与唯一的安全标识符结合使用。

在实际操作过程中，合约通过生成一个文档予以实现，该文档包含了基于法律格式的合约条款，以及所需的机器可读标签。同时，该文档由发行方及其私钥进行数字签名。随后，可采用消息摘要函数实现散列化，进而生成可以识别文档的哈希值。接下来，该哈希结果会在合约的执行过程中进一步使用和签署，以便将每项交易链接起来，同时采用标识符哈希值作为意图证据。图 6.1 显示了这一处理过程，通常称为蝴蝶结（bowtie）模型。

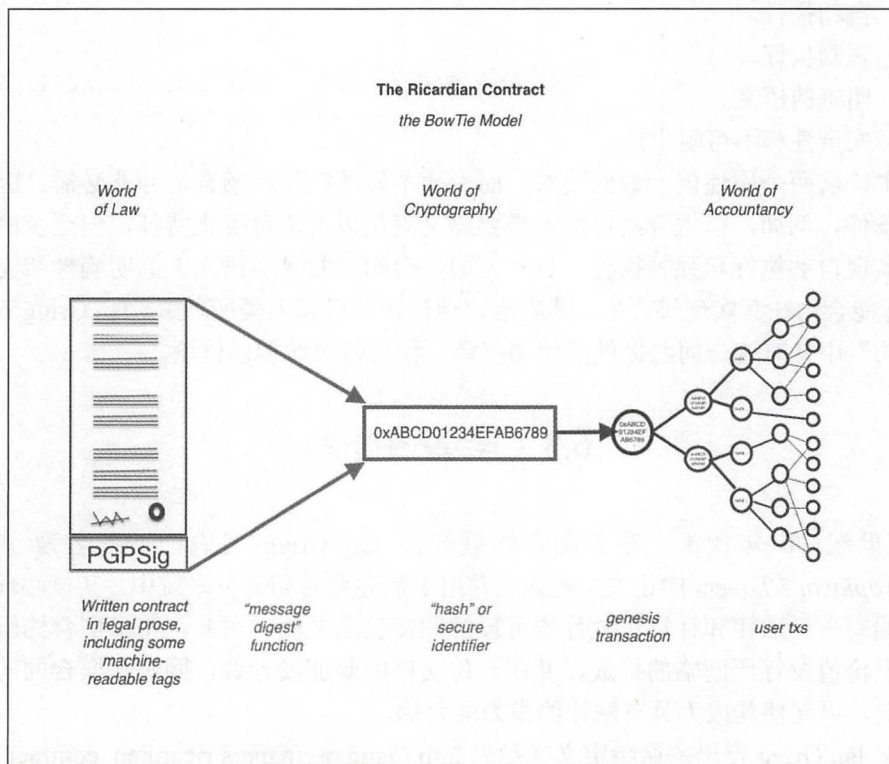


图 6.1 李嘉图合约中的蝴蝶结模型

图 6.1 中左侧表示文档起源于法律领域，并于随后实现了哈希化；而在会计领域，结果消息摘要则用作一个标识符。这里，会计领域基本上可以代表任意或多个会计、交易和信息系统，并被用于各项业务中，以执行各种业务操作。这一流程背后的思想可解释为，文档哈希化生成的消息摘要首先在创始交易中被使用（或者是第一项交易），随后在每项交易中作为一个标识符在整个合约的执行过程中使用。

通过这一方式，原始书面合约和会计领域的每一笔交易之间就建立了一个安全的链接。

李嘉图合约不同于智能合约，智能合约不包含任何合约文件，而且只专注于合约的执行。相比较而言，李嘉图合约则更关注于包含合约法律文本的、文档语义的丰富性和生成过程。合约的语义分为两种类型，即操作语义和指称语义。第一种类型定义了合约的实际执行、正确性和安全性，而后者则与全部合同的实际意义有关。对此，研究人员已经将智能合约代码和智能法律合约区分开来。其中，智能合约只涉及合同的执行；第二种合约则包含了法律协议的指称和操作语义。根据语义之间的不同，对智能合约进行分类具有实际意义。较好的方式是将智能合约看作是一个独立的实体，并可对法律条文和代码（业务逻辑）进行编码。在比特币上，可以看到一个较为简单的智能合约实现，且完全面向合约的执行过程；而李嘉图合约更倾向于生成一份人类可以理解的文档，计算机程序可理解其中的部分内容。这可以视作法律语义与操作性能（语义 vs 性能）之间的关系，如图 6.2 所示。这一观点是在 Ian Grigg 的论文 *On the intersection of Ricardian and smart contracts* 中提出的。

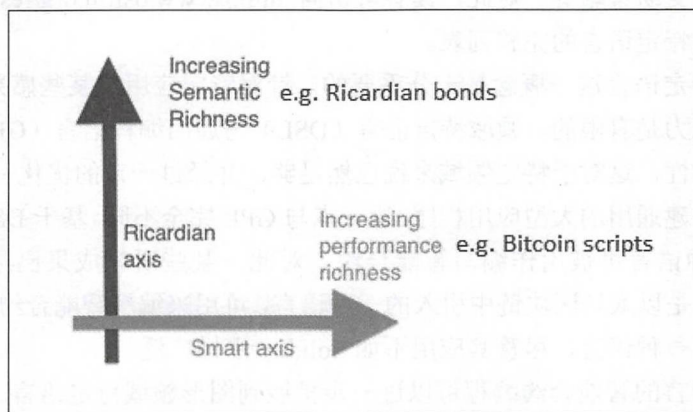


图 6.2 Ian Grigg 提出的性能与语义之间的正交关系。

本图对此稍作修改，以展示两个轴向上不同的合约类型

智能合约将上述两种元素（性能和语义）结合在一起，进而实现了一种理想模型。

李嘉图合约可以定义为 3 个对象的元组，即法律条文、参数和代码。其中，法律条文代表了常规语言制定的法律合约；代码则表示为计算机程序，即计算机可理解的的法律条文表达方式；参数将法律合约中的适宜内容关联到对应代码中。

李嘉图合约已经在多个系统中实现了，例如 CommonAccord、OpenBazaar、OpenBazaar 和 Askemos。

6.3.1 智能合约模板

智能合约适用于任何所需行业，当前，大多数用例均与金融行业相关。近期，金融行业提出了智能合约模板这一概念，其思想是构建标准模板，并提供一个框架来支持金融工具的法律协议。这一观点是 Clack 等人 在其论文 *Smart Contract Templates: Foundations, design landscape and research directions* 中提出的。除此之外，Clack 等人还建议，为了支持智能合约模板的设计和实现，还应设计特定于某种领域的语言。针对于此，一种名为 CLACK 的语言已于近期提到研发日程上来。该语言是一种用于增强型合约知识的通用语言，其用途非常丰富，并提供了多项功能，包括对法律条文的支持，以及多个平台和密码功能上的执行能力。

金融行业中的合约并非是一个全新的概念，各种特定领域内的语言（DSL）已经在金融行业中投入使用。例如，存在一些支持开发保险产品的 DSL，可代表能源衍生品，或者被用于构建交易策略等。对此，读者可访问 <http://www.dslfin.org/resources.html>，以了解金融领域内特定语言的完整列表。

理解领域特定语言这一概念是十分重要的。针对特定应用或某些感兴趣的领域，此类语言的表达能力是有限的。领域特定语言（DSL）与通用编程语言（GPL）不同：DSL 仅包含了少量特性，这对于特定领域来说已然足够，并经过一定的优化。另外，DSL 基本上不被用于构建通用的大型应用程序，这一点与 GPL 完全不同。基于 DSL 的设计理念，可以想象，这种语言可被用作编写智能合约。对此，某些研究成果已浮出水面，例如 Solidity。该语言是以太坊区块链中引入的一种语言，可用来编写智能合约。另外，Serpent 也是以太坊的另一种语言，尽管其应用不如 Solidity 那样广泛。

领域特定语言的智能合约编程可以进一步扩展到图形领域特定语言。在某个智能合约设计平台上，领域专家（而非程序员）可以采用图形用户界面和画布定义并绘制金融合约的语义和性能。一旦流程绘制完毕，即可进行先期测试，随后将其从系统中部署到目标平台上。其中，目标平台可以是区块链。同样，这也不是一个全新的概念，类似的

方案已用于 Tibco StreamBase 产品中。该方案基于 Java 系统，用于构建事件驱动的高频交易系统。

相关研究应在高级 DSL 开发领域的引导下进行，以便在用户友好的图形用户界面中设计智能合约，同时允许非程序员也加入到这一行列中来。

6.3.2 Oracle

Oracle 是智能合约生态系统的重要组成部分。智能合约的局限性在于无法访问外部数据，而这些数据可能需要控制业务逻辑的执行。例如，合同规定的证券的股票价格，以发放股息支付。针对于此，Oracle 可以用来为智能合约提供外部数据。Oracle 表示为一类接口，可将数据从外部源发送到智能合约。根据行业规范和实际需求，Oracle 可以提供不同类型的数据，包括天气预报、世界新闻，以及基于物联网设备数据的企业行为。Oracle 是一类可信的实体，可通过安全通道将数据传输至智能合约中。尽管 Oracle 具有一定的可信度，但在某些情况下，不当操控仍会产生错误数据。相关验证方案可以通过不同的公证制度予以提供，稍后将对此加以讨论。在该方法中，某些方法并非可取，即受信问题。那么，如何相信第三方提供的数据的质量和真实性？在金融领域该问题尤其明显，因为市场数据必须准确可靠。对于智能合约设计者来说，可接受一个信誉良好的第三方所提供的 Oracle 数据，但是集中化的问题仍然存在。这一类 Oracle 可称作标准 Oracle 或简单 Oracle。

鉴于去中心化特征，另一种类型的 Oracle 则称为去中心化 Oracle，并可以建立在分布式机制上。可以设想，Oracle 可以从另一个区块链中获取源数据，这是由分布式共识所驱动的，从而确保数据的真实性。例如，一家运行自身私链的机构可以通过 Oracle 发布数据，随后可被其他区块链所消耗。

除此之外，研究人员还引入了硬件 Oracle 这一概念，其中需要使用到源自物理设备的真实数据。例如，硬件 Oracle 可以用于遥测和物联网技术中。然而，该方案要求硬件设备不能被篡改。对此，可以采用防篡改设备以满足这一要求。

在现有的一些平台中，智能合约可通过 Oracle 获得外部数据。取决于所用的区块链类型，Oracle 使用不同的方法将数据写入区块链。例如，在比特币区块链中，Oracle 利用 OP_RETURN 操作码将数据写入到特定的交易中，而智能合约则可监视该交易并读取数据。一些在线服务提供了相应的 Oracle 服务，如 <http://www.oraclize.it/> 和 <https://www.realitykeys.com/>。除此之外，<https://smartcontract.com/> 同样可提供外部数据，以及智能合约支付能力。所有这些服务的目的在于，使智能合约获得执行和制定决策所

需的数据。对于 Oracle 从外部源获取的数据，为了证明其真实性，可采用 TLSnotary 这一类机制，进而生成数据源和 Oracle 间的通信证据。这可确保反馈至智能合约的数据从当前数据源中获得。读者可访问 <https://tlsnotary.org/>，以了解更多关于 TLSnotary 的细节内容。

图 6.3 显示了 Oracle 和智能合约生态圈之间的通用模型。

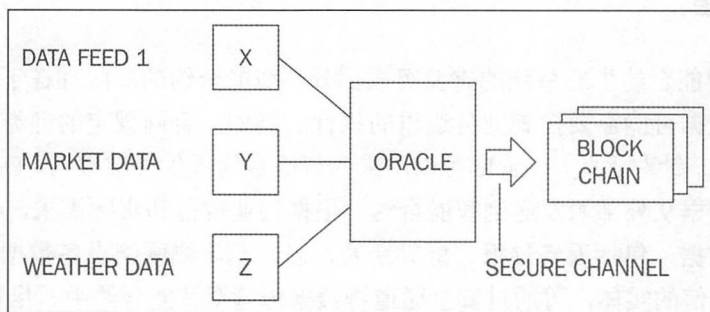


图 6.3 在区块链上，Oracle 和智能合约交互的简单模型

6.3.3 Smart Oracle

Codium 提出并实现了 Smart Oracle 这一理念。基本上讲，Smart Oracle 是一种与 Oracle 类似的实体，仅是添加了执行合约代码这一项功能。Codium 通过 Google Native Client 运行，并可视作一个沙箱环境，用于运行未受信任的 x86 本地代码。读者可访问 <https://www.codium.org/> 获取 Codium。

6.3.4 在区块链上发布智能合约

由于区块链提供的分布式共识机制，将智能合约部署到区块链上是十分有意义的。作为区块链的例子，以太坊支持开发和部署智能合约。Ethereum 区块链上的智能合约通常是大型应用程序中的一部分内容，例如去中心化自治组织（DAO）。

相比之下，在比特币区块链中，比特币交易的 `lock_time` 字段可视为智能合约基础版本的推动者。`lock_time` 字段使交易可以在指定时间或多个区块之后被锁定，从而执行一项基本合约，即在满足特定条件（经历的时间或区块数）的情况下，某项交易只能被解锁。然而，作为一类基本合约，从本质上讲其功能非常有限。除了上面提到的例子，比特币脚本语言也可用于构造基本的智能契约。同样，该语言的实际功能也具有较大的局限性。针对于此，一种可能性是，向一个比特币地址提供资金，任何人都可通过哈希冲

突攻击占用这笔资金。这一想法最初出现于 Bitcointalk 论坛中；另外，读者还可访问 <https://bitcointalk.org/index.php?topic=293382.0> 以了解更多信息。

6.3.5 DAO

DAO 是最高的众筹项目之一，并于 2016 年 4 月启动，基本上可视为一组智能合约，旨在提供一个投资平台。由于代码出现了 bug，2016 年 6 月曾被黑客入侵，相当于 5000 万美元的资产从 DAO 中转至另一个账户中。这导致以太坊中产生了一个硬分叉，以便从攻击中予以恢复。需要注意的是，此处的代码概念具有法律效应，或者是具有不可阻挡性的智能合约，但此类概念的实现尚缺乏应有的成熟度，不足以保证完全可信的信任机制。从这一类事件中可以看出，以太坊基金会能够通过引入一个硬分叉来阻止和改变 DAO 的行为。虽然这一硬分叉基于合理原因被引入，但依然违背了去中心化精神。另一方面，对硬分叉和某些矿工（继续在原始链上挖掘）的抵制导致了“以太坊经典”的诞生。“以太坊经典”表示原始的、非分叉的以太坊区块链。其中，代码仍然具有法律效应。这次攻击凸显了智能合约所面临的危险，且有必要针对智能合约开发一种正式的语言，同时也强调了测试的重要性。近期在以太坊中出现的各种漏洞，均围绕着智能合约开发语言而展开。因此，当务之急是制定一个标准框架以解决所有这些问题，前述内容已对此有所提及。针对智能合约语言的局限性问题，仍然需要更多研究成果的加入。

6.4 本章小结

本章首先介绍了智能合约的历史及其定义。由于尚未就智能合约的标准定义达成一致，本章尝试引入一个包含智能合约核心内容的定义。同时，本章还讨论了李嘉图合约，并解释了其与智能合约之间的区别。其中，李嘉图合约强调了合约的定义，而智能合约则关注合约的实际执行过程。此外，本章还阐述了智能合约模板这一概念，在学术界和业界，这一领域的研究十分活跃。接下来，本章探讨了创建领域特定语言的可能性，并在此基础上构建智能合约或智能合约模板。最后还引入了 Oracle 等概念，并简单地介绍了 DAO 和智能合约中的安全问题。

第 7 章 以太 坊

本章讨论以太坊区块链，包括其基本原理和某些更为高级的理论概念。另外，本章还将探讨与以太坊区块链相关的各种组件、协议和算法，以使读者理解区块链模式背后的理论基础。随后，本章还将介绍钱包软件、挖掘操作和设置以太坊节点的实际运作过程。同时，本章还将涉及一些挑战性话题，如以太坊面临的安全问题和可扩展性。此外，相关内容还将涉及到交易和市场动态。

7.1 简 介

2013 年 11 月，Vitalik Buterin 提出了以太坊这一概念，其关键思想是开发一种图灵完备的语言，同时支持任意区块链和去中心化应用程序（智能契约）的开发。这与比特币形成了鲜明对比，后者的脚本语言非常有限，仅支持基本和必要的操作。

7.1.1 以太坊客户端和发布

不同的以太坊客户端基本上采用了不同的语言进行开发，目前较为流行的是 go-Ethereum 和 Parity。其中，go-Ethereum 是基于 Golang 语言开发的，而 Parity 则通过 Rust 予以构造。除此之外，还存在其他的客户端。例如，对于大多数目标而言，geth（一种 go-Ethereum 客户端）已然足够。Mist 则是一类用户友好的图形用户界面（GUI）钱包，于后台运行 geth 并与网络同步。后续内容将对此加以详细介绍。

以太坊的第一个版本称作 Frontier；当前版本则命名为 homestead release；下一个版本将命名为 metropolis，并关注协议的简化和性能方面的改进；最终版本称作 serenity，并计划通过权益证明算法（Casper）予以实现。serenity 中其他领域的研究还包括可扩展性、隐私和以太坊虚拟机（EVM）的升级。考虑到这将是一项持续不断的开发工作，因而以太坊生态系统也将处于一个不断完善、发展的过程。因此，serenity 不应该被认为是最终的版本，而是一个漫长的、持续改进过程中的一个重要里程碑。除此之外，更多的版本也在构思中，只是尚未命名而已。与此相对应的是，Web 3.0 版本已被提至日程上来，在社区中也是一个被热议的话题。作为现有 Web 2.0 技术的进化，Web 3.0 强调语义和智能型 Web。其中，人、应用程序、数据和网络将连接在一起，并且能够以一种智能的方

式相互交流。随着区块链技术的出现，去中心化网络也逐渐浮出水面。实际上，这也是互联网的最初设想。也就是说，所有的主要服务，如 DNS、搜索引擎和互联网上的标识都将在 Web 3.0 中去中心化。对此，以太坊平台将有助于实现这一愿望。

7.1.2 以太坊栈

以太坊栈由各种组件组成，核心内容则是 P2P 以太坊网络上运行的以太坊区块链。其次，还包含一个在节点上运行的以太坊客户端（通常是 `geth`），并连接到 P2P 以太坊网络——区块链于此处下载并实现本地存储。该客户端提供了多项功能，包括挖掘和账户管理。另外，区块链的本地副本定期与网络同步。`web3.js` 库则是另一个组件，允许通过远程过程调用（RPC）接口与 `geth` 交互。

图 7.1 显示了以太坊栈中的各种组件。

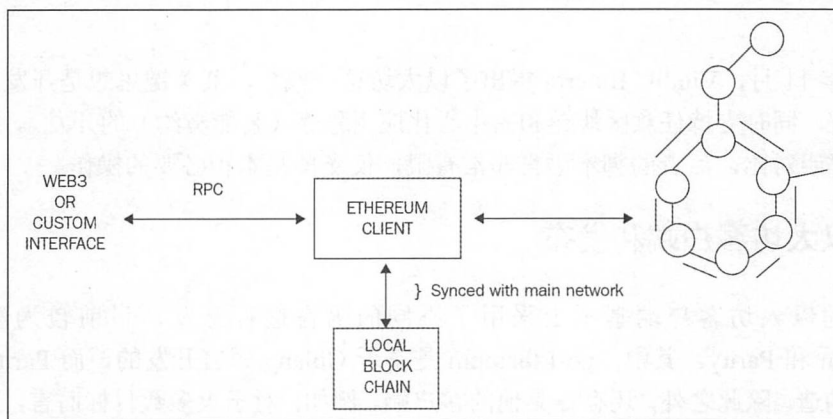


图 7.1 以太坊栈中的各种组件

7.2 以太坊区块链

与其他任何区块链一样，以太坊也可被可视化为基于交易的状态机。这一观点由 Gavin Wood 博士提出，其核心思想可描述为：当以增量方式执交易行事务时，初始状态将转换为最终状态。相应地，最终转换结果视为当前状态的终极版本。图 7.2 显示了以太坊状态转换函数，其中，某项交易事务的执行导致了状态转换。

在该例中，地址 4718bf7a 与 741F7A2 之间的两个 Ether 转移将被初始化。初始状态表示交易执行之前状态，而变换结果即为最终状态。后续内容还将对此加以更详细的讨

论, 当前示例旨在引入以太坊中状态转换这一核心概念。

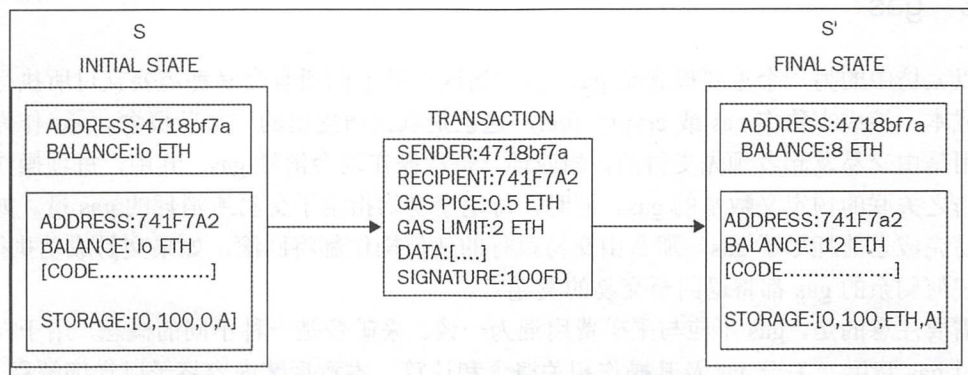


图 7.2 以太坊状态转换函数

7.2.1 货币 (ETH 和 ETC)

作为对矿工的激励, 以太坊提供了称为 **Ether** 的本地货币。在 DAO 被入侵之后 (稍后对此予以解释), 为了缓解该问题产生的负面影响, 此处产生了一个硬分叉 (分支)。因此, 当前存在两种以太坊区块链。其中, 以太坊经典, 其货币表示为 ETH, 而硬分叉版本则表示为 ETC, 且处于持续增长状态, 并且正在进行积极的开发。然而, ETC 也有自己的追随者, 以及一个处于壮大过程中的专业社区, 同时也是以太坊的“非分叉”原始版本。本章主要讨论 ETH, 这也是目前最活跃的官方以太坊区块链。

7.2.2 分叉

由于主协议升级, 目前最新发布了 **homestead** 版本, 从而产生了一个硬分叉。该协议在区块号 1150000 处升级, 因此, 以太坊从第一个版本 **Frontier** 迁移到第二个版本 **homestead**。

最近一次意外分叉发生在 2016 年 11 月 24 日, 时间是 14:12:07, 是由 **geth** 客户端日志记录机制中的一个错误所引发的。其中, 网络分叉出现于区块号 2686351 处。该错误导致 **geth** 在空 **out-of-gas** 异常的情况下无法恢复空账删除操作。相比之下, 这在 **Parity** (另一个流行的 **Ethereum** 客户端) 中并不构成问题。也就是说, 从区块编号 2686351 中, 以太坊区块链被分成两部分, 一部分随 **Parity** 客户端一起运行, 另一部分则与 **geth** 运行。因此, 这一问题随着 **geth** 版本 1.5.3 的发布而得到解决。

7.2.3 gas

以太坊中的另一个重要概念是 gas。以太坊区块链上的所有交易都需要支付所执行的计算成本，该成本称作 gas 或 crypto fuel，这也是以太坊提出的一个新概念。gas 作为交易费用是由交易发起者预先支付的。相应地，每次操作均会消耗 gas。其中，每项操作都包含与之关联的预定义数量的 gas。同时，每笔交易均指定了交易所消耗的 gas 量。如果在交易完成之前耗尽了 gas，那么由交易执行的任何操作都将回滚。如果交易成功执行，那么任何剩余的 gas 都将退回至交易的发起者。

需要注意的是，gas 不应与采矿费用混为一谈。采矿费是一种不同的概念，用于向矿工支付 gas 费用。关于 gas 及其操作相关概念和计算，本章后续内容将予以详细解释。

7.2.4 共识机制

以太坊的共识机制基于 GHOST 协议，该协议由 Sompolinsky 和 Zohar 于 2013 年 12 月首次提出。感兴趣的读者可访问 <http://eprint.iacr.org/2013/88.pdf> 查看原始论文。

以太坊使用了该协议的一个简化版本。其中，消耗计算量最大的构造链定义为最终版本。另一种观点是获取最长链——最长链必须通过消耗足够的开采量来建造。

对于快速生成时间所导致的旧区块和孤立区块，贪婪最重观察子树（Greedy Heaviest Observed Subtree, GHOST）最初是作为一种机制来缓解此类问题的。在 GHOST 中，旧区块被添加至计算中，进而获得最长的重链。在以太坊中，旧区块称作 Uncles 或 Ommers。

图 7.3 显示了最长链和重链之间的比较结果。

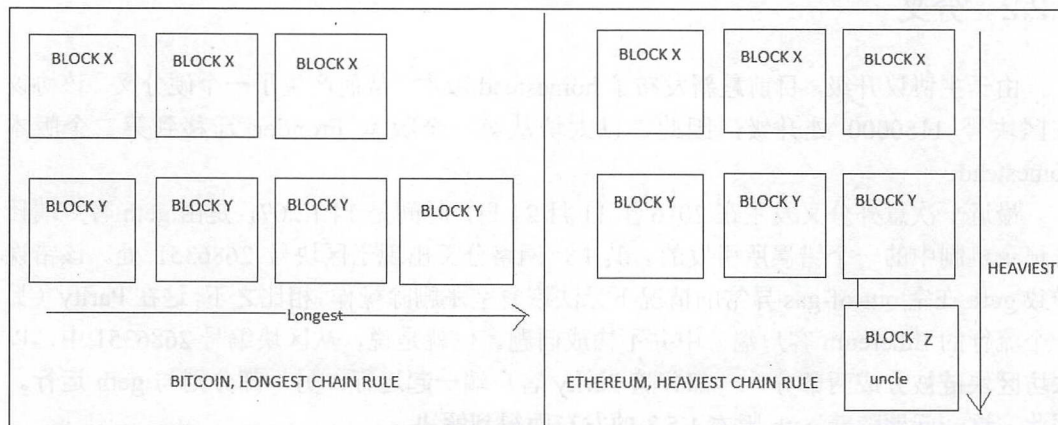


图 7.3 最长链和重链

7.2.5 世界状态

以太坊中的世界状态定义为以太坊区块链中的全局状态，基本上可表示为以太坊地址和账户状态之间的映射。其中，地址长度定义为 20 个字节。另外，映射行为表示为一种数据结构，并通过递归长度前缀（RLP）实现序列化。这里，RLP 是一种用于以太坊中的特定编码方案，对网络存储和传输执行二进制数据的序列化操作，并将最终状态存储于 Patricia 树中。RLP 函数将一个数据项作为输入，可以是一个字符串或一个数据项列表，并生成适合在网络上存储和传输的原始字节。RLP 并不对数据进行编码；相反，其主要目的是对结构进行编码。

账户状态由 4 个字段组成：nonce、balance、storageroot 和 codehash，下面将对此进行逐一解释。

- ❑ **nonce**：当每次从地址发送交易事务时，该值递增。在合约账户中，nonce 代表账户创建的合同数量。在以太坊中，合约账户是两种类型的账户之一，稍后将对此予以解释。
- ❑ **balance**：该值代表了 Wei 的数量。这是以太坊中对应地址所持有的最小货币单位（Ether）。
- ❑ **storageroot**：该字段代表 Merkle Patricia 树的根节点，并对账户的存储内容进行编码。
- ❑ **codehash**：作为一个不可变字段，包含了与账户关联的智能合约代码的哈希值。在普通账户中，该字段包含一个空字符串的 Keccak 256 位哈希值。另外，代码通过一个消息调用而被调用。

图 7.4 显示了全局状态及其与账户字典树、账户、区块头之间的关系。相应地，中图表示账户的数据结构，其中包含了源自账户存储字典树根节点（左图）的存储根哈希值。随后，账户数据结构用于全局状态字典树中，表示为地址和账户状态之间的映射。最后，通过 Keccak 256 位算法对全局状态字典树的根节点实现哈希化操作，并形成区块头数据结构中的部分内容，这在右图中作为状态根哈希值予以显示。

基本上讲，账户字典树定义为一棵 Merkle Patricia 树，用于对账户存储内容进行编码。对应内容存储为 keccak 256 位哈希值（隶属于 256 位整数键）和 RLP 编码的 256 位整数值之间的映射。

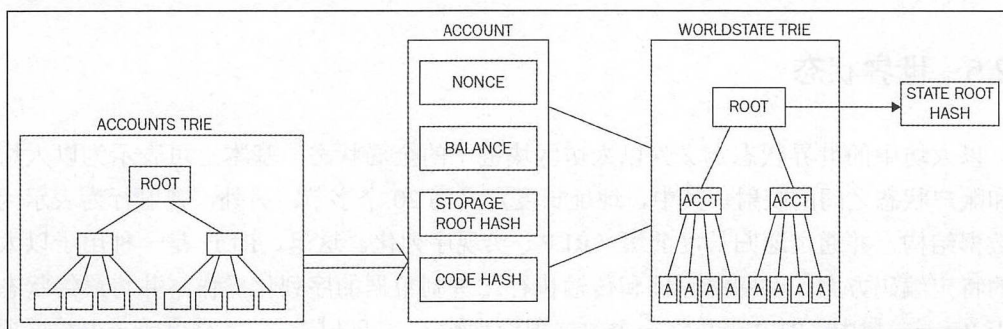


图 7.4 账户字典树（账户的存储内容）、账户元组、全局状态字典树、状态根哈希值之间的关系

7.2.6 交易

以太坊交易是可视作为数字签名的数据包，并使用了包含相关指令的私匙。当交易完成时，会产生消息调用或创建合约。根据所产生的输出，可以将交易划分为以下两种类型。

- ❑ 消息调用型交易：交易仅生成消息调用，用于将消息从一个账户传递到另一个账户。
- ❑ 合约生成型交易：顾名思义，此类交易导致新的合约产生。这也意味着，当交易成功执行后，将创建一个包含关联代码的账户。

上述两种类型的交易均包含了某些公共字段，如下所示：

(1) **nonce** 表示为数值，并在每次由发送者发送交易事务时递增。该值必须等于所发送的交易数量，同时作为交易的唯一标识符。注意，一个 **nonce** 值只能被使用一次。

(2) **gasPrice** 字段表示执行交易时所需的 Wei 数量。

(3) **gasLimit** 字段包含了最大 **gas** 值，并在进行交易时被消费。关于 **gas** 及其局限性，稍后还将进行详细讨论。就目前而言，该字段表示用户（例如，交易的发送者）愿意为计算支付的 Ether 费用。

(4) **to** 字段表示交易接收者的地址值。

(5) **Value** 表示为转移到接受者的 Wei 的总量。在合约账户中，该字段表示合约将持有的余额。

(6) **Signature** 由 3 个字段组成，分别是 **v**、**r** 和 **s**。对应值表示数字签名 (**R**, **S**)，以及可以用来恢复公钥 (**V**) 的信息；另外也可以由交易的发送方确定。**Signature** 基于 ECDSA 方案，并采用了 SECP256k1 曲线。关于椭圆曲线加密技术，第 3 章曾对此有所讨论。在本节中，ECDSA 将在其以太坊环境中予以呈现。

V 是一个单独的字节值，用以描述椭圆曲线点的大小和符号，该值可以是 27 或 28。另外，**V** 在 ECDSA 恢复合约中用作恢复 ID，该值可从私钥中恢复（获取）公钥。在

secp256k1 中, 恢复 ID 表示为 0 或 1。在以太坊环境下, 这将被 27 所抵消。关于 ECDSARECOVER 函数的更多细节, 本章后续内容还将对此加以深入分析。

R 源自曲线上的一个计算点。首先, 可选取一个随机数并与曲线生成器 (generator) 相乘, 进而计算曲线上的一点, 其 x 坐标表示为 R。同时, R 被编码为 32 字节序列。除此之外, R 必须大于 0 且小于 secp256k1n 限定值 (115792089237316195423570985008687907852837564279074904382605163141518161494337)。

S 的计算方式可描述为: 将 R 与私钥相乘并添加至签名消息散列中, 最终将其除以所选的随机数以计算 R。另外, S 也表示为 32 位字节序列。据此, R 和 S 共同确定了当前签名。

当签署交易时, 可使用 ECDSASIGN 函数。该函数将需要签名的消息和私钥作为输入, 并生成一个单字节值 V, 以及另一个 32 字节值 S。对应公式如下:

$$\text{ECDSASIGN}(\text{消息}, \text{私钥}) = (V, R, S)$$

(7) init 字段仅在创建合约的交易中使用, 表示为一个无限长度的字节数组, 并指定在账户初始化过程中使用的 EVM 代码。该字段中包含的代码仅执行一次, 此时当前账户首次被创建, 并在此之后立即被销毁。

init 还将返回另一个称为 body 的代码段。该代码段将继续运行, 以响应合约账户可能接收的消息调用。这一类消息调用可通过交易或内部代码予以发送。

(8) 如果交易表示为消息调用, 那么将使用 data 字段而不是 init, 并以此表示消息调用的输入数据。同样, 其尺寸不存在任何限制, 并以字节数组的形式加以组织。

在图 7.5 中, 交易表示为前述字段元组, 随后被纳入交易字典树 (即调整后的 Merkle-Patricia 树) 中。最后, 交易字典树的根节点通过 Keccak 256 位算法实现哈希化, 并与交易中的区块中的交易列表一同置入区块头中。

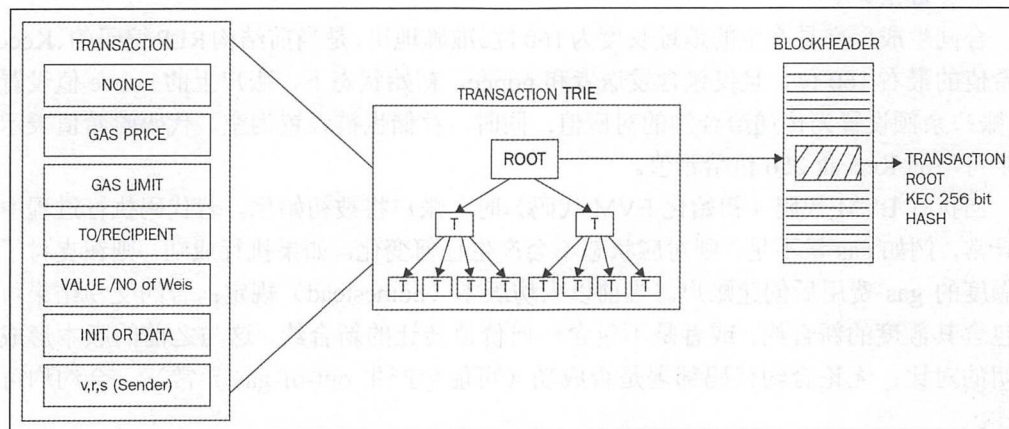


图 7.5 交易、交易字典树和区块头之间的关系

相应地，交易可在交易池或区块中获取。当某个挖掘节点执行区块验证操作时，将在交易池中最高支付交易处开始并逐一执行。当到达 gas 限定条件，或者交易池不存在需要处理的交易时，即开始挖掘过程。在该过程中，区块反复执行哈希操作，直至获取某个有效的 nonce。最终将生成一个小于难度目标的数值。

当到达 gas 限定条件，或者交易池不存在需要处理的交易时，即开始挖掘过程。在该过程中，区块反复执行哈希操作，直至获取某个有效的 nonce，最终将生成一个小于难度目标的数值。一旦区块被成功挖掘，将立即广播至网络并将被网络验证和接受。该过程类似于第 6 章所讨论的比特币的采矿过程。唯一的区别是，以太网的工作量证明算法具有抗 ASIC 性，即 Ethash，其中获取 nonce 需占用较大的内存空间。

7.2.7 合约生成型交易

当创建账户时，需要设置某些参数，如下所示：

- ☐ 发送者。
- ☐ 最初的交易者。
- ☐ 现有 gas 值。
- ☐ gas 价格。
- ☐ 初始状态下分配的捐赠额度。
- ☐ 包含任意长度的字节数组。
- ☐ 初始 EVM 代码。
- ☐ 消息调用/合约生成的当前堆栈深度（当前深度表示堆栈中已经存在的数据项数量）。

合同生成型交易产生的地址长度为 160 位。准确地讲，是当前结构 RLP 编码的、Keccak 哈希值的最右 160 位，且仅包含发送者和 nonce。初始状态下，账户上的 nonce 值设置为 0；账户余额设置为传递给合约的对应值；同时，存储也被设置为空。代码哈希值表示为空字符串的 Keccak 256 位哈希值。

当执行 EVM 代码（初始化 EVM 代码）时，账户将被初始化。若代码执行过程中出现异常，例如 gas 量不足，则对应状态不会产生任何变化。如果执行成功，则在支付了一定额度的 gas 费用后创建账户。当前以太坊版本（homestead）规定：合约交易结果可以是包含其额度的新合约，或者是不包含任何价值转让的新合约。这与之前的版本形成了鲜明的对比：无论合约代码部署是否成功（可能会产生 out-of-gas 异常），合约均可被创建。

7.2.8 消息调用型交易

消息调用需要使用到多个执行参数，如下所示：

- ☐ 发送者。
- ☐ 交易发起者。
- ☐ 接收者。
- ☐ 账户（其代码将被执行）。
- ☐ 现有 gas 量。
- ☐ 价值量。
- ☐ gas 价格。
- ☐ 任意长度的字节数组。
- ☐ 当前调用的输入数据。
- ☐ 消息调用/合约生成栈的当前深度。

消息调用将导致状态转换。另外，消息调用还生成输出数据——如果交易被执行，则不使用该数据。相应地，如果消息调用由 VM 代码触发，则使用交易执行过程中生成的输出结果。

图 7.6 显示了两种交易类型之间的隔离状态。

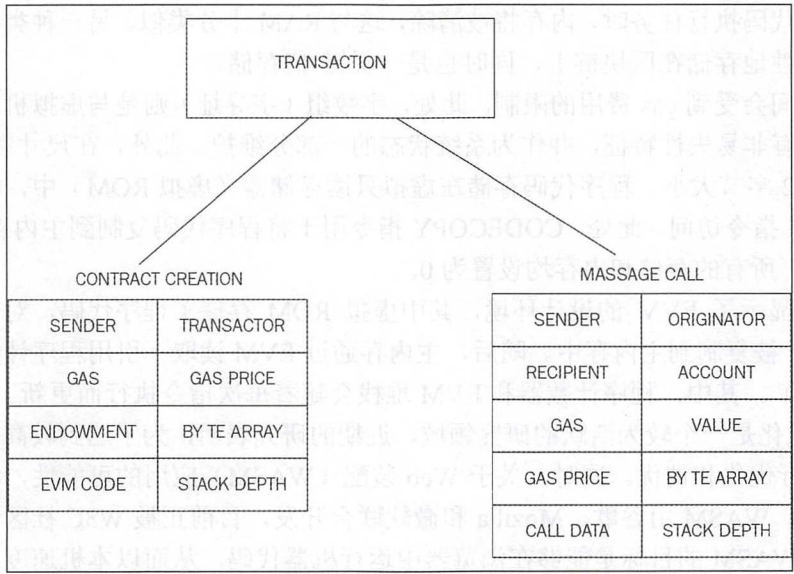


图 7.6 两种交易类型所需的执行参数

7.3 以太坊区块链中的元素

本节将介绍以太坊的各种组件和区块链。下面首先讨论 EVM 的基本概念。

7.3.1 以太坊虚拟机

EVM 是一种简单的基于堆栈的执行器，运行字节码指令，以便在系统状态间进行转换。其中，虚拟机的字大小设置为 256 位；栈大小则限制为 1024 个元素，并基于 LIFO（后进先出）队列。EVM 是一类图灵完备机，但会受到运行指令时所需的 gas 量的限制。这意味着，由于 gas 这一需求条件的限制，将不可能出现无限循环（可能导致拒绝服务攻击）。除此之外，EVM 还支持异常处理，例如 gas 量不足或无效的指令。在这种情况下，机器将立即停止并将错误返回至执行代理。

EVM 是一个完全隔离的沙箱运行环境。也就是说，在 EVM 上运行的代码不能访问任何外部资源，例如网络或文件系统。

如前所述，EVM 是一个基于堆栈的架构，且设计为 big-endian 格式，对应的字大小为 256 位宽，允许 Keccak 256 位哈希和椭圆曲线密码计算。

这里存在两种存储类型可用于合约和 EVM。类型一称作内存，并定义一个字节数组。当合约完成代码执行任务时，内存将被清除，这与 RAM 十分类似。另一种类型则称为存储，即永久性地存储在区块链上，同时也是一类键-值存储。

内存空间会受到 gas 费用的限制。此处，数组（字寻址）则是与虚拟机相关的存储结构，且具有非易失性特征，并作为系统状态的一部分维护。此外，在尺寸和存储方面，键-值均为 32 字节大小。程序代码存储在虚拟只读存储器（虚拟 ROM）中，并可以使用 CODECOPY 指令访问。此处，CODECOPY 指令用于将程序代码复制到主内存中。最初，在 EVM 中，所有的存储和内存均设置为 0。

图 7.7 显示了 EVM 的设计环境，其中虚拟 ROM 存储了程序代码，对应代码使用 CODECOPY 被复制到主内存中。随后，主内存通过 EVM 读取（引用程序计数器并逐步骤执行指令）。其中，程序计数器和 EVM 堆栈会随着每次指令执行而更新。

EVM 优化是一个较为活跃的研究领域，近期的研究表明，为了达到较高的性能，可对 EVM 进行优化和调优。同时，关于 Web 装配（WASM）应用的可能性，相关研究已经在进行中。WASM 由谷歌、Mozilla 和微软联合开发，目前正被 W3C 社区组织设计为开放标准。WASM 的目标是能够在浏览器中运行机器代码，从而以本机速度运行。类似地，EVM 2.0 的目标则是能够在 CPU 中以本地方式运行 EVM 指令集，从而提升运行速

度和效率。

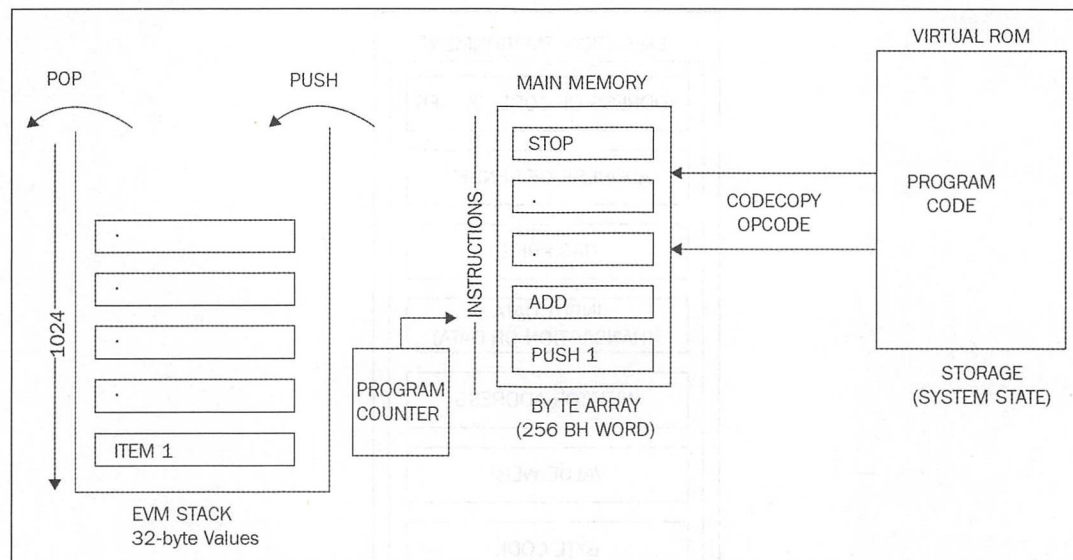


图 7.7 EVM 操作

7.3.2 执行环境

为了执行代码，执行环境需要提供一些关键元素。其中，核心参数由执行代理提供，例如某项交易，如下所示：

- ❑ 拥有执行代码的账户地址。
- ❑ 交易发送者地址，以及执行的最初地址。
- ❑ 交易中的 gas 价格，用于初始化当前执行过程。
- ❑ 基于执行代理类型的输入数据或交易数据，并定义为一个字节数组。在消息调用时，如果执行代理表示为某项交易，则交易数据作为输入数据。
- ❑ 账户地址，用于启动代码执行或交易的发送者。若代码执行由某项交易发起，则表示为发送者的地址；否则，则表示为账户地址。
- ❑ 值或交易值。表示为 Wei 数量。如果执行代理为某项交易，则表示为交易值。
- ❑ 所执行的代码呈现为字节数组形式（在每次执行循环中，由迭代函数获取）。
- ❑ 当前区块的区块头。
- ❑ 执行中消息调用或合约生成型交易的数量，即所执行的 CALL 或 CREATE 的数量。

在图 7.8 中，执行环境表示为包含 9 个元素的元组。

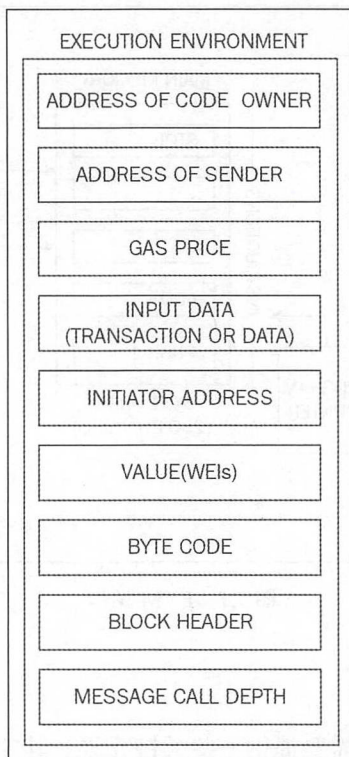


图 7.8 执行环境元组

除了前面提到的 9 个字段之外，系统状态和剩余 gas 也提供至执行环境。执行过程将产生结果状态、执行后剩余的 gas、自销毁或自消除设置（稍后将对此加以解释）、日志系列以及 gas 偿还额度。

1. 机器状态

机器状态也是通过 EVM 于内部维护的。在 EVM 的每个执行周期之后，都会更新机器状态。另外，迭代器函数（稍后将予以介绍）在虚拟机中运行，并输出状态机的单循环结果。相应地，机器状态表示为以下元素组成的元组：

- ❑ 现有 gas。
- ❑ 程序计数器，可表示最大值为 256 的正整数。
- ❑ 存储内容。
- ❑ 内存中活跃的字数量。

☐ 堆栈内容。

EVM 设计包含了异常处理，并在出现下列异常时终止执行：

☐ 执行中的 gas 量不足。

☐ 无效指令。

☐ 缺乏足够的堆栈数据项。

☐ jump 操作码的无效地址。

☐ 无效的堆栈尺寸（大于 1024）。

2. 迭代函数

前面提到的迭代函数执行各种重要功能，并以此设置机器的下一个状态，以及最终的全局状态。相关功能包括以下内容：

☐ 从字节数组中获取下一条指令，其中，机器码存储于执行环境中。

☐ 在堆栈中添加/删除（PUSH/POP）数据项。

☐ 根据指令/操作码的 gas 消费，递减 gas 值。

☐ 递增程序计数器（PC）。

机器状态可视为一个元组，如图 7.9 所示。

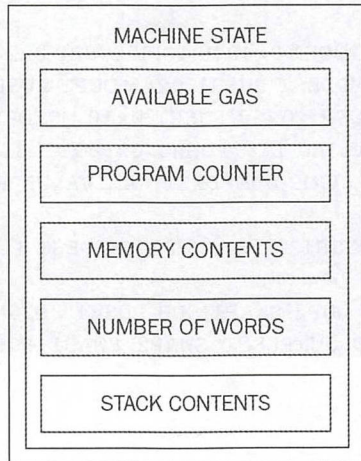


图 7.9 机器状态元组

如果在执行循环中遇到 STOP、SUICIDE 或 RETURN 操作码，虚拟机也可在正常条件下终止。

利用高级语言（如 Serpent、LLL 或 Solidity 语言）编写的代码可转换为 EVM 所理解的字节码，以便由 EVM 执行。Solidity 是一种高级语言，采用 JavaScript 开发（例如语法），

并可针对智能合约编写代码。代码编写完毕后，将被编译为字节代码以供 EVM 理解（使用称为 solc 的 Solidity 编译器）。

LLL（类似于 Lisp 的底层语言）则是编写智能合约代码的另一种语言；而 Serpent 语言则是一种与 Python 类似的高级语言，同样可针对以太坊编写智能合约。

例如，下列内容显示了简单的 Solidity 语言程序：

```
pragma solidity ^0.4.0;
contract Test1
{
    uint x=2;
    function addition1(uint x) returns (uint y) {
        y=x+2;
    }
}
```

该程序将转换为字节码。第 8 章将详细讨论 Solidity 语言的编译代码示例。当前示例对应的字节码如下：

```
606060405260e060020a6000350463989e17318114601c575b6000565b34600057
6029600435603b565b60408051918252519081900360200190f35b600281015b91
905056
Opcodes PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x2 PUSH1 0x0 SSTORE CALLVALUE
PUSH1 0x0 JUMPI JUMPDEST PUSH1 0x45 DUP1 PUSH1 0x1A PUSH1 0x0 CODECOPY
PUSH1 0x0 RETURN PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0xE0 PUSH1 0x2 EXP
PUSH1 0x0 CALLDATALOAD DIV PUSH4 0x989E1731 DUP2 EQ PUSH1 0x1C JUMPI
JUMPDEST PUSH1 0x0 JUMP JUMPDEST CALLVALUE PUSH1 0x0 JUMPI PUSH1 0x29
PUSH1
0x4 CALLDATALOAD PUSH1 0x3B JUMP JUMPDEST PUSH1 0x40 DUP1 MLOAD SWAP2
DUP3
MSTORE MLOAD SWAP1 DUP2 SWAP1 SUB PUSH1 0x20 ADD SWAP1 RETURN JUMPDEST
PUSH1 0x2 DUP2 ADD JUMPDEST SWAP2 SWAP1 POP JUMP
```

7.3.3 操作码及其含义

EVM 中定义了不同的操作码。相应地，根据所执行的操作，操作码可划分为不同类型，下面针对其含义和用途加以分析。

1. 算术运算

EVM 中的全部算术运算均为 2^{256} 模运算，并用于执行基本的算术运算。此类操作的对应值范围为 0x00~0x0b，如表 7.1 所示。

表 7.1 算术运算

助 记 符	值	POP	PUSH	Gas	描 述
STOP	0x00	0	0	0	停止执行
ADD	0x01	2	1	3	两个值的加法运算
MUL	0x02	2	1	5	两个值的乘法运算
SUB	0x03	2	1	3	减法运算
DIV	0x04	2	1	5	整数除法运算
SDIV	0x05	2	1	5	有符号除法运算
MOD	0x06	2	1	5	模-余数运算
SMOD	0x07	2	1	5	有符号模-余数运算
ADDMOD	0x08	2	1	8	模-加法运算
MULMOD	0x09	2	1	8	模-乘法运算
EXP	0x0a	2	1	10	指数运算（基数的乘法运算）
SIGNEXTEND	0x0b	2	1	5	扩展 2 的补码有符号整数

需要注意的是，STOP 并非算术运算操作，只是因为其值（0）位于当前范围内，因而被归入算术运算操作分类中。

2. 逻辑运算

逻辑运算用于执行比较和布尔逻辑操作，对应值位于 0x10~0x1a 范围内，如表 7.2 所示。

表 7.2 逻辑操作

助 记 符	值	POP	PUSH	Gas	描 述
LT	0x10	2	1	3	小于
GT	0x11	2	1	3	大于
SLT	0x12	2	1	3	有符号小于比较
SGT	0x13	2	1	3	有符号大于比较
EQ	0x14	2	1	3	等于比较
ISZERO	0x15	1	1	3	NOT 操作符
AND	0x16	2	1	3	位 AND 操作
OR	0x17	2	1	3	位 OR 操作
XOR	0x18	2	1	3	位抑或（XOR）操作
NOT	0x19	1	1	3	位 NOT 操作
BYTE	0x1a	2	1	3	从字中获取单字节

3. 密码操作

该分类中仅包含一项操作，即 SHA3，如表 7.3 所示。需要注意的是，这并非是 NIST 推出的标准化 SHA3，而是 Keccak 的原始实现。

表 7.3 密码操作

助 记 符	值	POP	PUSH	Gas	描 述
SHA3	0x20	2	1	30	用于计算 256 位 Keccak 哈希值

4. 环境信息

该分类中涵盖了 13 条指令，常用于生成与地址、运行期环境以及数据复制操作相关的信息，如表 7.4 所示。

表 7.4 环境信息

助 记 符	值	POP	PUSH	Gas	描 述
ADDRESS	0x30	0	1	2	获取当前执行账户的地址
BALANCE	0x31	1	1	20	获取既定账户的余额
ORIGIN	0x32	0	1	2	用于获取初始交易发送者的地址
CALLER	0x33	0	1	2	用于获取启动当前交易的账户地址
CALLVALUE	0x34	0	1	2	获取指令或交易发布的值
CALLDATALOAD	0x35	1	1	3	获取输入数据。该数据通过消息调用传递至某个参数中
CALLDATASIZE	0x36	0	1	2	用于复制利用消息调用所传递的输入数据尺寸
CALLDATACOPY	0x37	3	0	3	在当前环境与存储之间，用于复制利用消息调用传递的输入数据
CODESIZE	0x38	0	1	2	用于获取当前环境中运行代码的大小
CODECOPY	0x39	3	0	3	在当前环境和存储之间，复制运行代码
GASPRICE	0x3a	0	1	2	获取初始交易的 gas 价格
EXTCODESIZE	0x3b	1	1	20	获取特定账户代码的尺寸
EXTCODECOPY	0x3c	4	0	20	向存储中复制账户代码

5. 区块信息

该指令集与获取区块属性相关，如表 7.5 所示。

表 7.5 区块信息

助 记 符	值	POP	PUSH	Gas	描 述
BLOCKHASH	0x40	1	1	20	针对近期的 256 个完整区块，获取某个区块的地址
COINBASE	0x41	0	1	2	在区块中，获取受益者的地址
TIMESTAMP	0x42	0	1	2	获取区块中的时间戳
NUMBER	0x43	0	1	2	获取区块号
DIFFICULTY	0x44	0	1	2	获取区块难度
GASLIMIT	0x45	0	1	2	获取区块的 gas 限定值

6. 堆栈、内存和流操作

相关指令如表 7.6 所示。

表 7.6 堆栈、内存和流操作

助 记 符	值	POP	PUSH	Gas	描 述
POP	0x50	1	0	2	从堆栈中移除数据项
MLOAD	0x51	1	1	3	从内存中载入一个字
MSTORE	0x52	2	0	3	向内存中保存一个字
MSTORE8	0x53	2	0	3	向内存中保存一个字节
SLOAD	0x54	1	1	50	从内存中载入一个字
SSTORE	0x55	2	0	0	向内存中保存一个字
JUMP	0x56	1	0	8	调整程序计数器
JUMPI	0x57	2	0	10	根据某项条件调整程序计数器
PC	0x58	0	1	2	在递增之前获取程序计数器中的数值
MSIZE	0x59	0	1	2	获取活动内存的尺寸(以字节计)
GAS	0x5a	0	1	2	获取现有的 gas 量
JUMPDEST	0x5b	0	0	1	标记有效的跃址地址，且不会对执行中的机器状态产生任何影响

7. Push 操作

Push 操作用于将数据项置于堆栈中。对应的指令范围定义为 0x60~0x7f。EVM 中包含了 32 条指令，如表 7.7 所示。Push 操作将从程序代码的字节数组中读取。

表 7.7 Push 操作

助 记 符	值	POP	PUSH	Gas	描 述
PUSH1...PUSH32	0x60~0x7f	0	1	3	在堆栈中设置 N 个右对齐的 big-endian 字节数据项。其中，N 表示为基于助记符的、范围为 1~32 字节的数值

8. 复制操作

顾名思义，复制操作用于复制堆栈中的数据项。对应值的范围表示为 0x80~0x8f，如表 7.8 所示。相应地，EVM 中包含了 16 DUP 指令。另外，从堆栈中设置或移除数据项在助记符上也会体现出递增变化。例如，DUP1 从堆栈中移除某一项，并在堆栈中设置两项数据；相应地，DUP 16 则从堆栈中移除 16 项并放置 17 项。

表 7.8 复制操作

助 记 符	值	POP	PUSH	Gas	描 述
DUP1...DUP16	0x80~0x8f	X	Y	3	用于复制堆栈中的第 N 项。其中，N 表示为 DUP 指令号码；X 和 Y 则分别表示为堆栈中移除或置入的数据项

9. 交换操作

SWAP 操作提供了堆栈数据项的交换功能，同时存在 16 条 SWAP 指令。根据操作码类型，利用每条指令，堆栈数据项将被移除或置入（多达 17 项），如表 7.9 所示。

表 7.9 交换操作

助 记 符	值	POP	PUSH	Gas	描 述
SWAP1...SWAP16	0x90~0x9f	X	Y	3	用于交换堆栈中的第 N 项。其中，N 表示为 SWAP 指令号码；X 和 Y 则分别表示为堆栈中移除或置入的数据项

10. 日志操作

日志操作提供了相关操作码，并将日志项添加至子状态元组的日志系列字段中。

表 7.10 显示了 4 项日志操作，对应值范围表示为 0x0a~0xa4。

表 7.10 日志操作

助 记 符	值	POP	PUSH	Gas	描 述
LOG0...LOG4	0x0a~0xa4	X	Y(0)	375,750,1125, 1500,1875	用于添加包含 N 项内容的日志记录。其中，N 表示为 LOG 操作码号码。例如，LOG0 表示未包含任何内容的日志记录；LOG4 表示包含 4 项内容的日志记录。另外，X 和 Y 分别表示堆栈中移除和置入的数据项。根据所采用的 LOG 操作，X 和 Y 在 (2,0)~(6,0) 范围内呈递增变化

11. 系统操作

系统操作用于执行各种与系统相关的指令，例如创建账户、消息调用以及执行控制。表 7.11 显示了 6 种操作码。

表 7.11 系统操作

助 记 符	值	POP	PUSH	Gas	描 述
CREATE	0xf0	3	1	32000	用于创建包含关联代码的新账户
CALL	0xf1	7	1	40	用于将消息调用初始化为一个账户
CALLCODE	0xf2	7	1	40	利用备用账户代码将消息调用初始化至当前账户
RETURN	0xf3	2	0	0	终止执行并返回输出信息
DELEGATECALL	0xf4	6	1	40	等同于 CALLCODE，但不改变发送者的当前值
SUICIDE	0xff	1	0	0	终止当前执行，注册当前账户以供后续删除使用

在以太坊 homestead 版本中的 EVM 中，共计 129 个操作码，至此已基本介绍完毕。

7.4 预编译合同

以太坊中包含 4 种预编译合同，下面将对其予以介绍。

7.4.1 椭圆曲线公钥恢复函数

ECDSARECOVER（椭圆曲线 DSA 恢复功能）在地址 1 可用，并表示为 ECREC，且执行过程需要 3000 个 gas。如果签名无效，则该函数不会返回输出结果。公钥恢复是一种标准机制，可以从椭圆曲线密码中的私钥生成公钥。

ECDSA 恢复函数如下：

$$\text{ECDSARECOVER}(H, V, R, S) = \text{公钥}$$

该函数需要 4 项输入：H（签名消息的 32 字节哈希值）、V、R 和 S，代表 ECDSA 签名和恢复 ID，并生成一个 64 字节的公钥。本章前述内容已对 V、R 和 S 有所讨论。

7.4.2 SHA256 位哈希函数

SHA256 位哈希函数是一个预编译的合约并在地址 2 可用，同时生成输入内容的 SHA256 哈希值。该函数类似于传递函数。对 SHA256 的 gas 需求取决于输入数据的大小。相应地，输出结果则是一个 32 字节的数值。

7.4.3 RIPEMD160 位哈希函数

在地址 3 中，可以使用 RIPEMD160 位哈希函数提供 RIPEMD160 位哈希值。该函数的输出结果是一个 20 字节的数值。与 SHA256 类似，gas 需求依赖于输入数据的数量。

7.4.4 恒等函数

恒等函数在地址 4 中可用，并表示为 ID。该函数只是将输出定义为输入；换句话说，无论输入的 ID 函数是什么，恒等函数都会输出相同值。另外，gas 需求通过如下简单公式计算： $15 + 3[I_d/32]$ 。其中， I_d 表示输入数据。这意味着，在较高层次上，gas 需求取决于输入数据的大小，尽管这其中涉及了某些计算，如前述方程式所示。

上述预编译合约均可形成本地扩展，未来有可能纳入至 EVM 操作码中。



7.5 账 户

账户是以太坊区块链的主要构建块之一。由于账户之间的交互，状态被创建或更新。账户以及账户之间执行的操作体现了状态转换。这里，状态转换通过所谓的以太坊转换函数实现，其工作原理如下：

- ❑ 通过语法检测、签名有效性以及 `nonce` 确认交易的有效性。
- ❑ 计算交易费用，并通过签名确定发送地址。此外，还需检测发送者的账户余额并减去该余额，同时递增 `nonce` 值。如果账户余额不足，则返回错误。
- ❑ 提供足够的 `Ether` (`gas` 价格) 以支付交易成本。对应值根据交易的大小按字节支付。
- ❑ 该步骤将产生实际的价值转移。相应地，流量从发送者的账户传至接收者的账户。如果交易中指定的目标账户尚不存在，则自动创建账户。
- ❑ 此外，如果目标账户表示为合约，则执行相应的合约代码。除此之外，该过程还取决于可用的 `gas` 量。如果 `gas` 足够，合约代码将被完全执行；否则，流程将会跳至 `gas` 耗尽点。

若账户余额或 `gas` 不足，则交易失败。除了支付给矿工的费用外，所有状态变化都将回滚。

- ❑ 最后，剩余费用（若存在）返回至发送者；作为变更，相关费用支付给矿工。此时，函数返回最终状态。

以太坊中存在两种账户类型，包括外部持有账户和智能合约。

外部持有账户简称 EOA，类似于比特币中私钥控制的账户。合约账户包含了与私钥相关联的代码。相应地，EOA 中包含了 `Ether` 余额，能够发送交易且不存在关联代码；而合约账户（CA）中则涵盖了余额和关联代码，以及在响应交易事务或消息时触发和执行代码的能力。值得注意的是，由于以太坊区块链的图灵完备属性，合约账户内的代码包含了任意复杂度。该代码由 EVM 在以太坊网络上的每个挖掘节点执行。此外，合约账户可维持自身参数状态，并可以调用其他合约。可以想象，在 `Serenity` 版本中，外部持有账户和合约账户之间的差别可能被消除。

7.6 区 块

如前所述，区块是区块链中的主要构成部分，以太坊区块中包含了多种组件，其中



包括区块头、交易列表以及 Ommers 或 Uncles 头列表。

其中，交易列表仅是区块中所含的全部交易列表。此外，该区块还包括了 Uncles 头列表。相比较而言，最重要和最复杂的部分是区块头，下面将对此加以讨论。

7.6.1 区块头

区块头是以太坊区块中最为重要的组件，其中涵盖了大量的细节内容和相关信息。

- ❑ 父哈希值：表示为父（前一）区块头的 Keccak 256 位哈希值。
- ❑ Ommers 哈希值：包含在当前区块中的、Ommers（Uncles）区块表的 Keccak 256 位哈希值。
- ❑ 受益者：该字段包含了接收者的 160 位地址。若区块被成功挖掘，则受益者获得采矿奖励。
- ❑ 状态根节点：状态根节点字段包含了交易字典树根节点的 Keccak 256 位哈希值，并在全部交易处理完毕后进行计算。
- ❑ 交易根节点：表示为交易字典树根节点的 Keccak 256 位哈希值。其中，交易字典树代表了区块中的交易列表。
- ❑ 收据根节点：表示交易收据字典树根节点的 Keccak 256 位哈希值。其中，字典树由区块中所有交易的收据组成，每项交易处理后将生成交易收据，并包含了交易结束后的相关信息。更多细节内容稍后将作讨论。
- ❑ 日志过滤器：即 Bloom 过滤器，由日志地址和日志主题构成，此类内容源自区块交易列表中各项交易收据的日志条目，稍后将对此予以介绍。
- ❑ 难度：指当前区块的难度级别。
- ❑ 前区块数量：顾名思义，表示前区块的全部数量。其中，创始区块定义为 0 区块。
- ❑ gas 限制：该字段表示每个区块所限定的 gas 消费值。
- ❑ gas 消费量：表示区块中交易所消费的全部 gas 量。
- ❑ 时间戳：表示区块初始化时的时间。
- ❑ 附加数据：用于存储与区块相关的附加数据。
- ❑ Mixhash：该字段包含一个 256 位的哈希值，当与 nonce 结合使用时，即可证明已经花费了足够的计算量来创建区块。
- ❑ nonce：表示一个 64 位哈希值（一个数字），当与 mixhash 字段相结合使用时，用于证明为了创建区块，已经花费了足够的计算工作量。

图 7.10 显示了区块和区块头的详细结构。

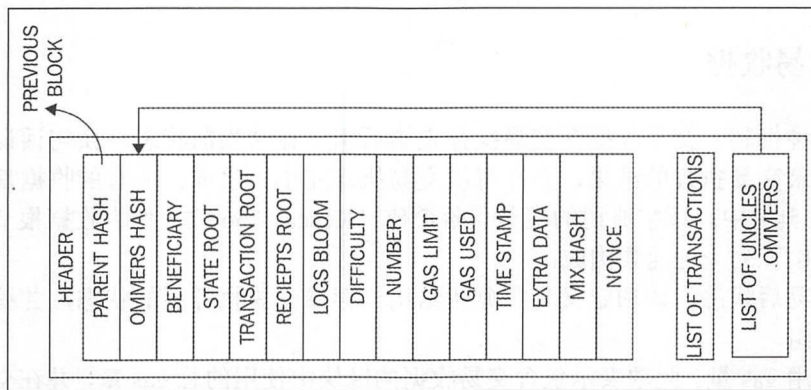


图 7.10 包含区块头的区块结构

7.6.2 创始区块

相对于所包含的数据以及常规区块的创建方式，创始区块则稍有不同，其中包含了 15 项内容，如表 7.12 所示。

表 7.12 创始区块中的 15 项元素。数据源自 Etherscan.io 提供的实际版本

元 素	描 述
时间戳	Jul-30-2015 03:26:13 PM +UTC
交易	8893 项交易以及 0 合约内部交易
哈希值	0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3
父哈希值	0x00
Sha3Uncles	0x1dcc4de8dec75d7aab85b567b6ccdd41ad312451b948a7413f0a142fd40d49347
挖掘	15 秒内 0x00
难度	17179869184
整体难度	17179869184
尺寸	540 字节
gas 限制	5000
所消费的 gas	0
nounce	0x00000000000000042
区块奖励	5 Ether
Uncles 奖励	0
附加数据	»èÛN4{NOE"} fpäúi3³ÛiËÛz8ää ,ú (Hex:0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbdb7a38e1e50b1b82fa)



7.6.3 交易收据

作为一种机制，交易收据在交易执行完毕后用于存储当前状态。换句话说，此类结构被用来记录交易执行的结果，并在每次交易执行完毕后生成。所有的收据都存储在一个索引-键字典树中。该字典树的根节点哈希值（Keccak 256 位）作为收据根节点置于区块头中，并由下列 4 种元素构成：

- ❑ 交易后状态。该项定义为字典树结构，存储交易执行后的状态，并编码为字节数组。
- ❑ 所用 gas 量。此项表示包含交易收据的区块中使用的总 gas 量，并在交易执行完毕后立即使用该值。注意，所用的总 gas 量应是一个非负整数。
- ❑ 日志集合。该字段显示了作为交易执行结果所创建的日志项集合。其中，日志项包含日志记录器的地址、一系列日志主题和日志数据。
- ❑ Bloom 过滤器。Bloom 过滤器是由日志集合中所包含的信息创建的。这里，日志条目被简化为一个 256 字节的哈希值，然后将其作为日志 Bloom 嵌入至区块头中。相应地，日志条目由日志记录器的地址、日志主题和日志数据组成。其中，日志主题被编码为一系列 32 字节的数据结构。最后，日志数据由几个字节的数据组成。

上述处理过程如图 7.11 所示。

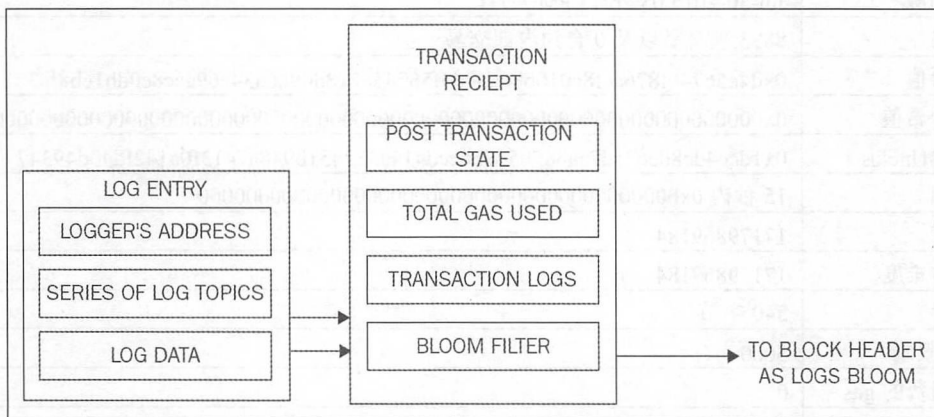


图 7.11 交易收据和 Bloom 过滤器

7.6.4 交易验证和执行

交易一般在其有效性验证完毕后开始执行，初始测试包含以下内容：



- ❑ 交易必须是格式良好的 RLP 编码，且不包含任何附加的尾随字节。
- ❑ 应确保交易签名的有效性。
- ❑ 交易 nonce 须等于发送者账户中的当前 nonce。
- ❑ gas 限定条件不应小于交易所消费的 gas。
- ❑ 发送者账户中应包含足够的余额以支付执行费用。

交易子状态在交易执行过程中被创建，并在交易结束后即刻被执行。此处，交易子状态表示为一个 3 元组。

- ❑ 销毁集合。该元素包含了交易执行完毕后须处理掉的账户。
- ❑ 日志系列。表示索引化的检查点系列，允许监视和通知合约调用以太坊环境外部的实体，例如应用程序前端。其工作原理类似于每次调用特定函数或发生特定事件时执行的触发器机制。另外，日志的创建旨在响应智能合约中产生的事件，同时也可以作为一种更简单的存储方式。关于交易的操作示例，第 8 章将对此加以讨论。
- ❑ 退还余额。表示启动执行过程的、交易中的 gas 总价格。注意，退还余额并不立即执行；相反，它们被用来部分抵消全部执行成本。

图 7.12 显示了交易子状态元组。



图 7.12 交易子状态

7.6.5 区块验证机制

若通过下列测试，则以太坊区块视为有效：

- ❑ 与 Uncles 和交易保持一致。也就是说，全部 Ommers (Uncles) 符合 Uncles 属



性，且工作量证明处于有效状态。

- ❑ 若前（父）区块存在且有效。
- ❑ 区块的时间戳有效。这基本上意味着，当前区块的时间戳必须大于父块的时间戳。而且小于 15 分钟。另外，所有区块的时间都是按 UNIX 时间计算的。

如果上述检测中的任意一条无效，则对应区块将被剔除。

1. 区块终止

区块终止是一个由矿工运行的过程，用来验证区块中的内容并实施奖励。其中包含以下 4 个步骤：

- ❑ Ommers 验证（无效区块也称为 Uncles）。也就是说，在采矿时确定 Ommers。无效区块的验证过程将检查区块头是否有效，且 Uncles 与当前区块之间的关系是否满足最大深度（6 个区块）。另外，一个区块最多可以包含两个 Uncles。
- ❑ 交易验证。在挖掘时确认交易，该过程在最终交易后，检测在区块中消耗的总 gas 是否等于最终的 gas 消耗。
- ❑ 奖励机制。通过回报（奖励）余额更新受益者的账户。在以太坊中，对矿工的奖励也涉及无效区块，约占当前区块奖励的 1/32。相应地，当前区块中的 Uncles 将获得全部区块奖励的 7/8。其中，当前区块奖励是 5 Ether。另外，一个区块最多包含两个 Uncles。
- ❑ 状态和 nonce 验证。在挖掘过程中，计算有效状态和 nonce。

2. 区块难度

如果两个区块之间的时间减少，则区块难度增加；若两个区块间的时间增加，则难度降低。对此，应维护一个大约一致的区块生成时间。在以太坊的 homestead 版本中，难度调整算法如下：

```
block_diff = parent_diff + parent_diff // 2048 *  
max(1 - (block_timestamp - parent_timestamp) // 10, -99) +  
int(2**((block.number // 100000) - 2))
```

上述算法表明，在区块的生成过程中，如果父区块与当前区块的时间差小于 10 秒，则难度就会增加；如果时间差在 10~19 秒之间，难度则保持不变；最后，如果时间差是 20 秒或更多，难度就会降低。相应地，这种下降与时间差是成正比的。

需要注意的是，在上述算法的最后一行代码中，每隔 100000 个区块难度将呈指数级增加，这也称作以太坊网络的难度时间炸弹或冰河世纪。在未来的某个时间点中，将难以实现以太坊区块链中的挖掘工作。鉴于 POW 链上的挖掘过程最终会变得非常困难，这将鼓励用户转至权益证明方案。根据该算法的最新更新和测试结果，在 2017 年下半年和



2021 年，区块生成时间将会显著提高，以至于无法在 POW 链上挖掘。这样，矿工们将别无选择，只能转至以太坊所提供的、称为 Casper 的权益证明方案。

7.7 Ether

对于所耗费的计算，Ether 可视为一种货币奖励，并通过交易和区块验证确保网络安全。Ether 用在以太坊区块链中，用于在 EVM 上支付合约的执行。除此之外，Ether 还用于购买 gas，进而执行以太坊区块链上的相关计算。

表 7.13 显示了相应的额度表。

表 7.13 额度表

单 位	Wei 值	Weis
Wei	1 Wei	1
Babbage	1e3 Wei	1000
Lovelace	1e6 Wei	1000000
Shannon	1e9 Wei	1000000000
Szabo	1e12 Wei	1000000000000
Finney	1e15 Wei	1000000000000000
Ether	1e18 Wei	1000000000000000000

费用将针对 EVM 执行的每项计算收取，稍后介绍具体的收费计划。

7.7.1 gas

针对以太坊区块链上执行的各项操作，须支付 gas。考虑到 EVM 的图灵完备特征，该机制可确保无限循环不会导致整体区块链处于停顿状态。交易费用则作为一定量的 Ether 被收取，该项费用取自交易发起者的账户余额。相应地，交易费用由挖掘过程中的矿工支付。如果这笔费用过低，则交易难以被矿工发现；费用越高，相关交易被矿工发现的概率也就越大。最终，若包含相应支付费用的交易被矿工纳入区块中，但涵盖了大量的复杂操作，在 gas 额度不足时将导致 out-of-gas 异常。此时，交易无效，但仍会形成当前区块中的部分内容，但交易发起者不会收到任何退额。

这里，交易付费成本如下式所示。

$$\text{全部成本} = \text{gasUsed} \times \text{gasPrice}$$



其中, gasUsed 表示执行过程中被交易消费的总 gas 量; 而 gasPrice 则由交易发起者指定, 以激励矿工将当前交易纳入下一个区块中。gasPrice 以 Ether 计, 且各 EVM 操作码均包含了一项分配费用。当然, 这仅是一项估算结果——与交易发起者最初指定的额度相比, 实际消费 gas 可能有所增减。例如, 如果计算耗时过长, 或者智能合约的行为由于其他一些因素而发生变化, 那么, 交易执行过程中操作量的变化也将导致 gas 消费量出现变化。如果执行过程中 gas 被耗尽, 则一切事物立即回滚; 如果执行成功且尚余少量 gas, 则将其返回至交易发起者。

每项操作均会消费一定量的 gas, 表 7.14 显示了一些操作的收费方式。

表 7.14 gas 付费方式

操 作 名 称	gas 消费	操 作 名 称	gas 消费
步进	1	sload	20
停止	0	txdata	5
销毁	0	交易	500
SHA3	30	生成合约	53000

根据表 7.14 以及前述公式, SHA3 操作计算示例如下所示:

- ❑ SHA3 消费 30 个 gas。
- ❑ 当前货币价格为 25 GWei, 即 0.000000025 Ether。
- ❑ 二者相乘有 $0.000000025 \times 30 = 0.00000075$ Ether。

因此, 全部 gas 最终收取 0.00000075 Ether。

7.7.2 费用标准

作为操作执行的先决条件, 以下 3 种情况下将实现 gas 付费:

- ❑ 执行某项操作计算。
- ❑ 生成合约或消息调用。
- ❑ 增加内存开销。

前述内容已对基于 gas 的各项指令和操作有所讨论。

7.8 消 息

消息表示为在两个账户间传递的数据和相关值。实际上, 消息定义为传递于两个账



户间的数据包，该数据包中包含了数据和数值（Ether 数量）。对此，数据包可通过智能合约（自治对象）或外部操作者（外部持有账户）发送，并以发送者数字签名的交易形式出现。

合约可向其他合约发送消息。此处，消息存在于执行环境中且未被存储。另外，消息类似于交易，二者间的主要差别在于：消息由合约生成，而交易则通过以太坊环境外部实体（外部持有账户）生成。

消息由下列组件构成：

- ☐ 消息发送者。
- ☐ 消息收据。
- ☐ 发送的 Wei 数量，以及合约地址消息。
- ☐ 可选的数据字段（合约的输入数据）。
- ☐ 可消费的最大 gas 量。

当合约执行 CALL 或 DELEGATECALL 操作码时，将生成消息。

调用行为不会向区块链广播任何内容，且仅是合约函数的本地调用，并在该节点本地运行。尽管与本地函数调用十分相近，但并不消耗任何 gas——调用行为仅是一项只读操作。另外，调用仅在本地执行，通常不会导致任何状态更改。正如在黄皮书中定义的那样，调用仅是账户间的消息传递行为。若目标账户包含了相关的 EVM 代码，那么虚拟机将从接收消息开始执行所需的操作。如果消息发送者是一个自治对象，则当前调用将传递从虚拟机操作返回的任何数据。

相应地，状态可被交易修改，这主要源自外部因素，消息经签名后即传播至以太坊网络上。

7.9 挖 掘

新货币可通过挖掘处理添加至区块链中，同时也可视为针对矿工的奖励行为，以验证和确认构成交易的相关区块。通过对计算进行验证，挖掘处理可确保网络的安全性。

在理论层面上，矿工将执行以下各项功能：

- ☐ 监听以太坊上的交易广播，进而确定所处理的交易。
- ☐ 确定 Uncles 或 Ommers 无效区块，并将其纳入至当前区块中。
- ☐ 利用区块挖掘获得的奖励更新账户余额。
- ☐ 最后，计算有效状态并结束当前区块，这将定义全部状态交易的最终结果。

当前挖掘方法基于工作量证明，且与比特币类似。若某一个区块视为有效，则不仅

满足一般性的一致性要求，还需针对既定难度包含工作量证明。

随着 serenity 版本的发布，工作量证明算法已被权益证明算法所替代。为了制定适用于以太坊网络的权益证明算法，大量的研究工作已付诸实施。

Casper 算法经推出后即有替换现有以太坊工作量证明算法之势，并可视为基于经济协议的保证金。其中，须通过节点在生成区块前放置一笔保证金。因此，在 Casper 中，节点命名为担保验证器，而保证金的置入行为则称作担保操作。

7.9.1 Ethash

Ethash 表示为以太坊中工作量证明的名称，最初被命名为 Dagger-Hashimoto 算法。自首次实现以来，大量内容已发生了变化，PoW 算法最终形成了当前人们所知的 Ethash。类似于比特币，挖掘背后的核心理念是获取一个 nonce（预定义难度级别中的哈希结果）。初始状态下，难度级别相对较低，在一定程度上，甚至 CPU 和独立 GPU 挖掘即可获得利润；当前，情况则有所转变，即采用矿池或大型 GPU 挖掘处理过程。

Ethash 是一类 memory-hard 算法，难以在专门的硬件上实现。类似于比特币，长期以来 ASIC 导致了挖掘中心化问题，而基于 memory-hard 算法的工作量证明则可缓解这一问题。同样，以太坊实现了 Ethash 并以此解决 ASIC 问题。根据 nonce 和区块头，该算法需要选择称为 DAG（有向无环图）的固定资源子集。其中，DAG 的尺寸约为 2GB，且每 30000 个区块发生变化。挖掘节点首次启动并生成 DAG 时，挖掘过程方才启动。另外，每 30000 个区块之间的时间约为 5.2 天，也称作周期。基于名为 Ethash 的工作量证明，该 DAG 可用作种子。根据当前规范，这一周期时间定义为 30000 个区块。

若成功获得一个有效 nonce，当前奖励方案为 5 个 Ether。除了收获 5 个 Ether 之外，矿工还将得到区块中所消耗的 gas 成本，以及区块中纳入无效块（Uncles）时的额外奖励。另外，每个区块中最多允许出现两个 Uncle，并得到常规区块奖励的 7/8。为了实现 12 秒的区块时间，区块难度可在每个区块上进行调整。相应地，对应奖励正比于矿工的哈希速率，这基本上体现了矿工的哈希计算速度。

通过简单地加入以太坊网络，并运行相应的客户端，即可执行挖掘操作。此处较为关键的要求是，在开始挖掘之前，节点应该与主网络完全同步。

下面将讨论挖掘过程中的各种方法。

7.9.2 CPU 挖掘

尽管在主网络上难以获利，但在测试网络甚至是私有网络中，CPU 挖掘依然具有一

定价值,进而可尝试进行挖掘和合约部署。第8章将讨论私有网络和测试网络的实际例子。此处将考察一个 geth 示例,以展示如何启动 CPU 挖掘。对此,可利用 mine 启动 geth,进而开始挖掘过程,如下所示:

```
geth --mine --minerthreads <n>
```

除此之外,还可以使用 web 3 geth 控制台启动 CPU 挖掘。另外, geth 控制台可以通过以下命令开启:

```
geth attach
```

在此之后,可以通过下列命令启动矿工。如果成功,则返回 true,否则将返回 false。

```
Miner.start(4)  
True
```

上述命令将利用 4 个线程启动矿工;而下列命令则终止矿工行为,若成功则返回 true。

```
Miner.stop  
True
```

7.9.3 GPU 挖掘

基本上讲, GPU 挖掘可通过下列命令执行:

```
geth --rpc
```

一旦 geth 启动并运行,区块链将被完全下载,随后即可运行 Ethminer 以开始挖掘。这里, Ethminer 是一个独立的矿工,可用于 farm 模式下,以对矿池发挥作用。用户可访问 <https://github.com/Genoil/cpp-ethereum/tree/master/releases> 下载 Ethminer,如下所示:

```
ethminer -G
```

使用 G 开关运行时,假定正确地安装并配置了相应的显卡。如果未发现合适的显卡, Ethminer 将返回一个错误,如图 7.13 所示。

```
drequinox@drequinox-OP7010:~$ ethminer -G  
[OPENCL]:No OpenCL platforms found  
No GPU device with sufficient memory was found. Can't GPU mine. Remove the -G argument  
drequinox@drequinox-OP7010:~$
```

图 7.13 未发现 GPU 时返回的错误信息

GPU 挖掘需要使用到 AMD 或 NVidia 显卡以及相应的 OpenCL SDK。对于 NVidia

芯片，用户可访问 <https://developer.nvidia.com/cuda-downloads> 下载；对于 AMD 芯片，用户可访问 <http://developer.amd.com/tools-and-sdks/rocm-zone/amd-accelerated-parallel-processing-app-sdk> 下载。

当图形卡安装、配置完毕后，可通过 `ethminer -G` 命令启动处理过程。

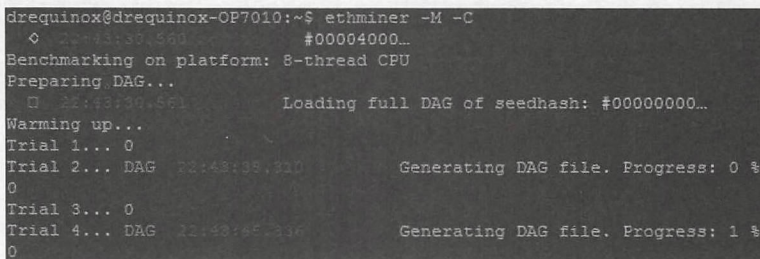
Ethminer 也可以用来运行基准测试。对此，存在两种基准测试模式，即 CPU 基准测试，如下所示：

```
$ ethminer -M -C
```

以及 GPU 基准测试，如下所示：

```
$ ethminer -M -G
```

图 7.14 显示了 CPU 挖掘基准测试。



```
drequinox@drequinox-OP7010:~$ ethminer -M -C
Q 22:43:30.561 #00004000...
Benchmarking on platform: 8-thread CPU
Preparing DAG...
Q 22:43:30.661 Loading full DAG of seedhash: #00000000...
Warming up...
Trial 1... 0
Trial 2... DAG 22:43:35.320 Generating DAG file. Progress: 0 %
0
Trial 3... 0
Trial 4... DAG 22:43:45.316 Generating DAG file. Progress: 1 %
0
```

图 7.14 CPU 基准测试

相应地，所用的 GPU 设备可通过下列命令予以确定：

```
$ ethminer -M -G --opencl-device 1
```

由于 GPU 挖掘通过 OpenCL AMD 实现，因而基于芯片组的 GPU 比 NVidia GPU 的工作速度更快。另外，考虑到高内存需求（DAG 构建），FPGA 和 ASIC 与 GPU 相比并不占优势，旨在阻止开发专门用于采矿的硬件。

7.9.4 挖掘设备

随着时间的推移，Ether 挖掘的难度越来越大，矿工们开始构建包含多个 GPU 的挖掘设备。一般情况下，挖掘设备通常包含 5 个 GPU，并以并行方式挖掘，从而提升获取有效 nonce 的概率。

用户可自行构建挖掘设施，或者从供应商处购买。下面将讨论典型的挖掘设备配置

组件。

1. 主板

用户需要购买一块配置多个 PCI-E x1 或 x16 插槽的专业主板，例如，BIOSTAR HIFI 或 ASRock H81。

2. SSD 硬盘驱动器

此处推荐使用 SSD 硬盘，其性能比类似设备更加高效。SSD 硬盘驱动器主要用于存储区块链。

3. GPU

GPU 是设备的重要组成部分，也是挖掘操作的主要平台，例如，包含 4GB RAM 的 Sapphire AMD Radeon R9 380 图形卡。

在操作系统方面，通常可选取 Linux Ubuntu 的最新版本作为操作平台。除此之外，另一种可用的 Linux 版本 EthOS (<http://ethosdistro.com/>) 则是专门为以太坊而建立的，并支持采矿操作。

最后，一些挖掘软件同样不可或缺，例如 Ethminer 和 geth 等。此外，还可安装某些远程监控和管理软件，以便在需要时可以远程监控和管理设备。注意，多个 GPU 在运行过程中会释放大量的热量（如图 7.15 所示），因此，现场应配置良好的空调制冷设施。注意，还需安装相应的监控软件以便针对硬件问题提示用户，例如 GPU 过热。

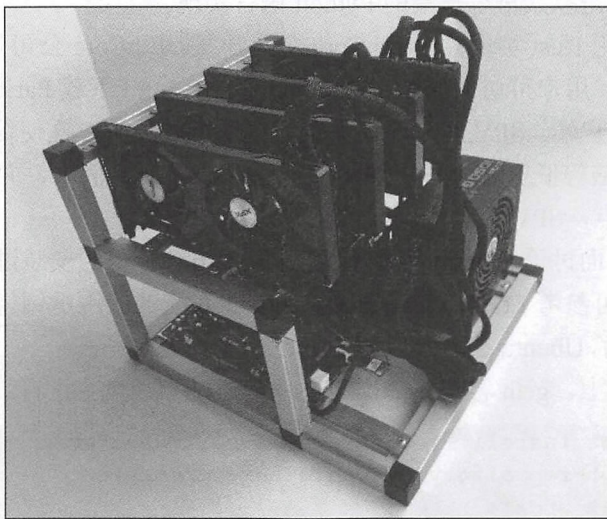


图 7.15 以太坊挖掘设备

4. 矿池

一些在线挖掘池提供了以太坊挖掘功能。据此，以太坊矿工可以使用相关命令连接到矿池。其中，各种矿池均提供了自己的指令，但一般情况下，连接矿池的过程基本类似。下列命令展示了源自 `ethereumpool.co` 中的一个示例：

```
ethminer -C -F
http://ethereumpool.co/?miner=0.1@0x024a20cc5feba7f3dc3776075b3e60c20eb1459
c@DrEquinox
```

对应结果如图 7.16 所示。

```
drequinox@drequinox-OP7010:~$ ethminer -C -F http://ethereumpool.co/?miner=0.1@0x024a20cc5feba7f3dc3776075b3e60c20eb1459c@DrEquinox
miner
Getting work package...
```

图 7.16 以太坊矿工

7.10 客户端和矿工

随着以太坊的发展和演化，在过去的几年中涌现出了大量的组件、客户端和工具。下面列出了某些主要组件、客户端软件和以太坊工具，旨在了解其用途和重要性。

- (1) `geth` 是以太坊客户端的 Go 语言实现。
- (2) `Eth` 是以太坊客户端的 C++ 语言实现。
- (3) `Pyethapp` 是以太坊客户端的 Python 语言实现。

(4) Parity 采用 Rust 实现并由 EthCore 公司开发。EthCore 公司是一家致力于 Parity 客户端开发的公司，用户可访问 <https://ethcore.io/parity.html> 下载 Parity。

(5) 轻量级客户端。SPV 客户端仅下载了区块链的子集，这使得诸如移动电话、嵌入式设备或平板电脑等低资源设备能够验证交易。在当前示例中，并不需要一个完整的以太坊区块链和节点，SPV 客户端仍然可以验证交易的执行。

(6) 安装。下面讨论 Ubuntu 系统上各种以太坊客户端的安装过程。关于其他操作系统的指令，读者可参考 Ethereum Wikis。Ubuntu 系统的应用示例将在后续内容中加以介绍，当前仅描述了 Ubuntu 的安装过程。

在 Ubuntu 系统上，`geth` 客户端的安装过程可通过下列命令进行：

```
> sudo apt-get install -y software-properties-common
> sudo add-apt-repository -y ppa:ethereum/ethereum
> sudo apt-get update
> sudo apt-get install -y ethereum
```

待安装完毕后，通过简单的 `geth` 命令即可启动 `geth`，其中先期配置了全部所需参数，进而可连接至以太坊网络（主干网），如下所示：

```
> geth
```

- ❑ **Eth 安装。** Eth 是以太坊客户端的 C++ 语言实现，并可通过下列命令在 Ubuntu 上进行安装：

```
> sudo apt-get install cpp-ethereum
```

- ❑ **Mist 浏览器。** Mist 浏览器是一个用户友好的界面，向终端用户提供了丰富的功能，用于浏览 DAPPS、管理账户以及合约。第 8 章将介绍 Mist 浏览器的安装。

当首次启动 Mist 时，将于后台初始化 `geth` 并与网络同步，如图 7.17 所示。根据不同网络速度和类型，网络同步过程可能会占用几个小时或几天。当采用 TestNet 时，同步的完成速度相对较快——TestNet (Ropsten) 的尺寸远小于 MainNet。第 8 章将讨论与 TestNet 连接相关的更多信息。

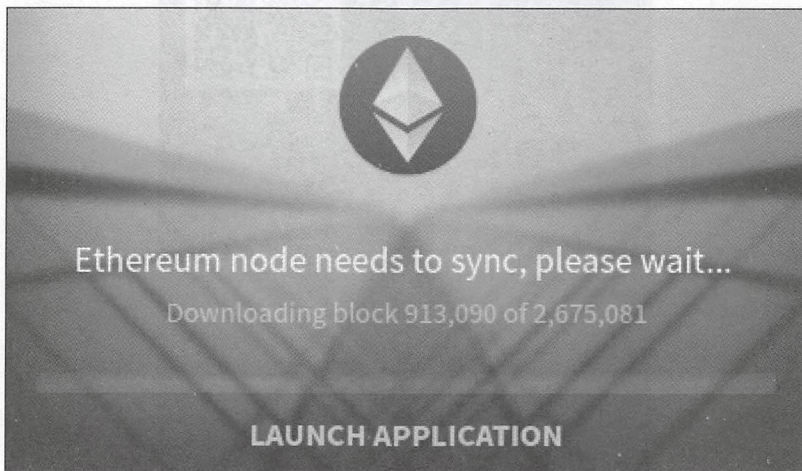


图 7.17 启动 Mist 浏览器并与主网络同步

需要说明的是，Mist 浏览器并非是钱包软件，而是一款 DAPPS 浏览器，为创建和管理合约、账户，以及浏览去中心化的应用程序提供了用户友好的界面。相应地，以太坊钱包则是一款随 Mist 发布的 DAPP。

钱包是一类通用程序，基于存储于其中的地址，可以存储私钥和关联账户。另外，它可以通过查询区块链来计算与地址关联的现有 Ether 余额。

其他钱包软件还包括（但不限于）MyEtherWallet，该软件是采用 JavaScript 进行开发

的开源 Ether 钱包，并可在客户端浏览器中运行。对应网址为 <https://www.myetherwallet.com>。

Icebox 由 Consensys 开发并推出，该浏览器可提供 Ether 的安全存储。当然，这取决于 Icebox 所运行的计算机设备是否已连接至互联网。

以太坊钱包软件涵盖了桌面级、移动端以及 Web 平台等多个版本。图 7.18 显示了较为流行的以太坊 iOS 钱包软件 Jaxx。

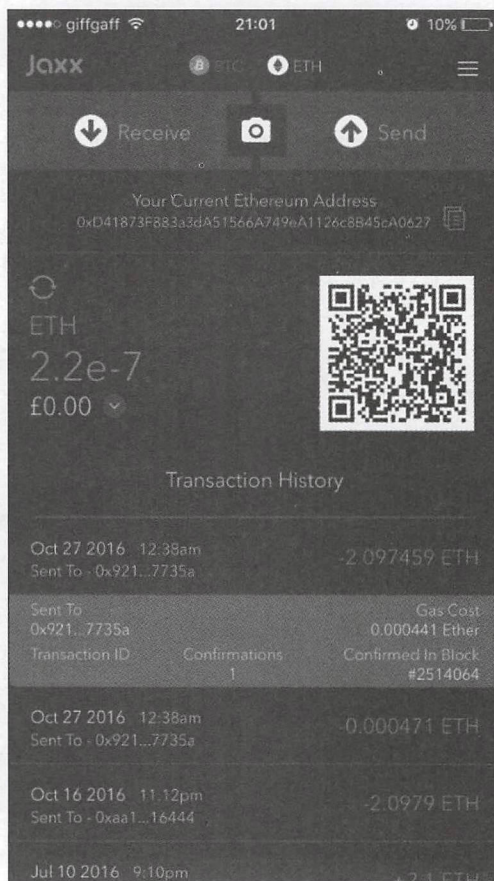


图 7.18 iOS 平台上的以太坊钱包 Jaxx，其中显示了当前交易和余额

若区块链同步完毕，Mist 将显示如图 7.19 所示的界面。在当前示例中，显示了 4 个账户且不包含任何余额。

新账户包含多种创建方式。如图 7.20 所示，在 Mist 浏览器中，可选择 Accounts→New account 命令；或者在 Mist Accounts Overview 界面中选择 Add account 选项。

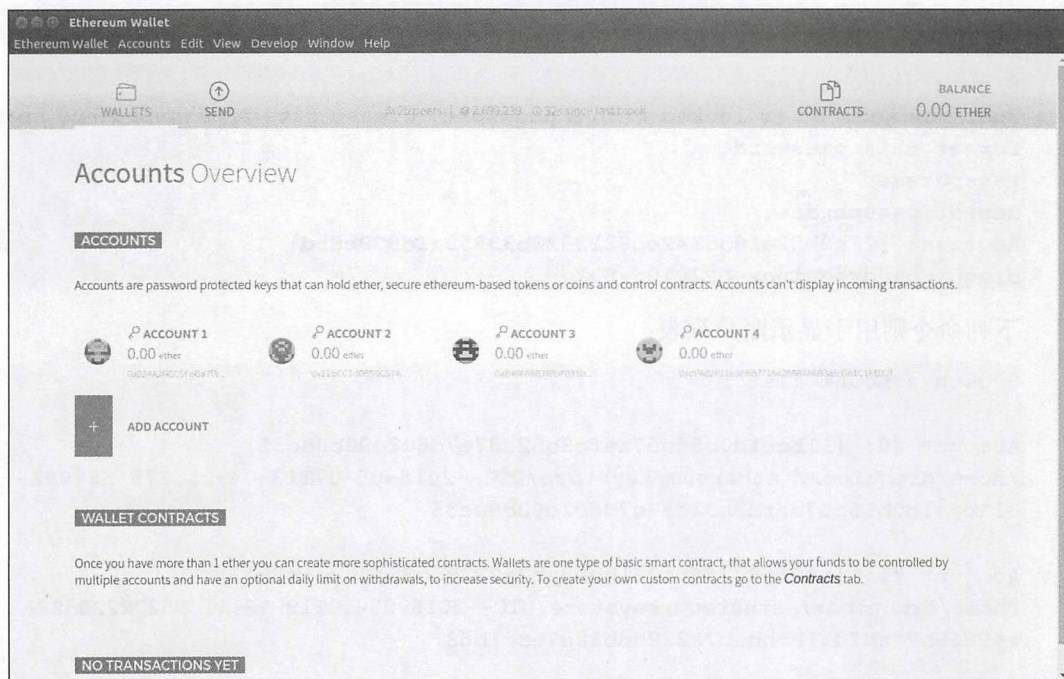


图 7.19 Mist 浏览器

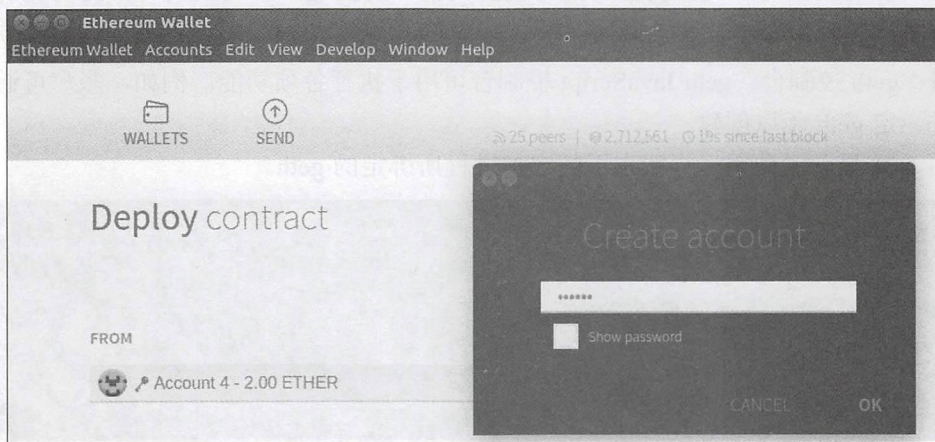


图 7.20 添加新账户

其中，相应账户需要设置密码。当账户设置完毕后，即显示于 Mist 浏览器中的账户浏览部分中。

除此之外，账户还可通过命令行进行添加，例如 `geth` 或 `parity`，稍后将对此讨论。

❑ geth。下列命令将创建新账户：

```
$ geth account new
Your new account is locked with a password. Please give a password. Do not
forget this password.
Passphrase:
Repeat passphrase:
Address: {21c2b52e18353a2cc8223322b33559c1d900c85d}
drequinox@drequinox-OP7010:~$
```

下列命令则用于显示账户列表：

```
$ geth account list

Account #0: {11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec35}
/home/drequinox/.ethereum/keystore/UTC--2016-05-07T13-04-15.175558799Z-
-11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec35

Account #1: {e49668b7ffbf031bbbdab7a222bdb38e7e3e1b63}
/home/drequinox/.ethereum/keystore/UTC--2016-05-10T19-16-11.952722205Z-
-e49668b7ffbf031bbbdab7a222bdb38e7e3e1b63

Account #2: {21c2b52e18353a2cc8223322b33559c1d900c85d}
/home/drequinox/.ethereum/keystore/UTC--2016-11-29T22-48-09.825971090Z-
-21c2b52e18353a2cc8223322b33559c1d900c85d
```

❑ geth 控制台。geth JavaScript 控制台可用于执行各项功能。例如，账户可通过绑定 geth 予以创建。

图 7.21 显示了利用运行状态下的守护进程所绑定的 geth。

```
drequinox@drequinox-OP7010:~$ geth attach
Welcome to the Geth JavaScript console!

instance: Parity//v1.4.4-beta-a68d52c-20161118/x86_64-linux-gnu/rustc1.13.0
coinbase: 0x0000000000000000000000000000000000000000000000000000000000000000
at block: 2718377 (Tue, 29 Nov 2016 22:52:52 GMT)
modules: eth:1.0 net:1.0 parity:1.0 parity_accounts:1.0 personal:1.0 rpc:1.0 traces:1.0 web3:1.0

> █
```

图 7.21 绑定 geth

当 geth 与以太坊客户端（当前为 Parity）的运行实例成功绑定后，将显示命令提示符“>”，进而提供了交互式命令行界面，并可通过 JavaScript 标记与以太坊客户端交互。

例如，在 geth 控制台中，可利用下列命令添加新账户：

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xc64a728a67ba67048b9c160ec39bacc5626761ce"
>
```

类似地，账户列表可通过下列命令加以显示：

```
> eth.accounts
["0x024a20cc5feba7f3dc3776075b3e60c20eb1459c",
"0x11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec35",
"0xdf482f11e3fbb7716e2868786b3afede1c1fb37f",
"0xe49668b7ffbf031bbbdab7a222bdb38e7e3e1b63",
"0xf9834defb35d24c5a61a5fe745149e9470282495"]
```

❑ 向账户提供比特币。该选项随 Mist 浏览器而推出。对此，可单击当前账户并选择该选项向账户中注入资金。另外，shapeshift.io 则是支持该操作的后台引擎，并可用于向当前账户注入比特币或其他货币，例如法定货币。

当兑换完毕后，转让的 Ether 即出现于当前账户中，如图 7.22 所示。

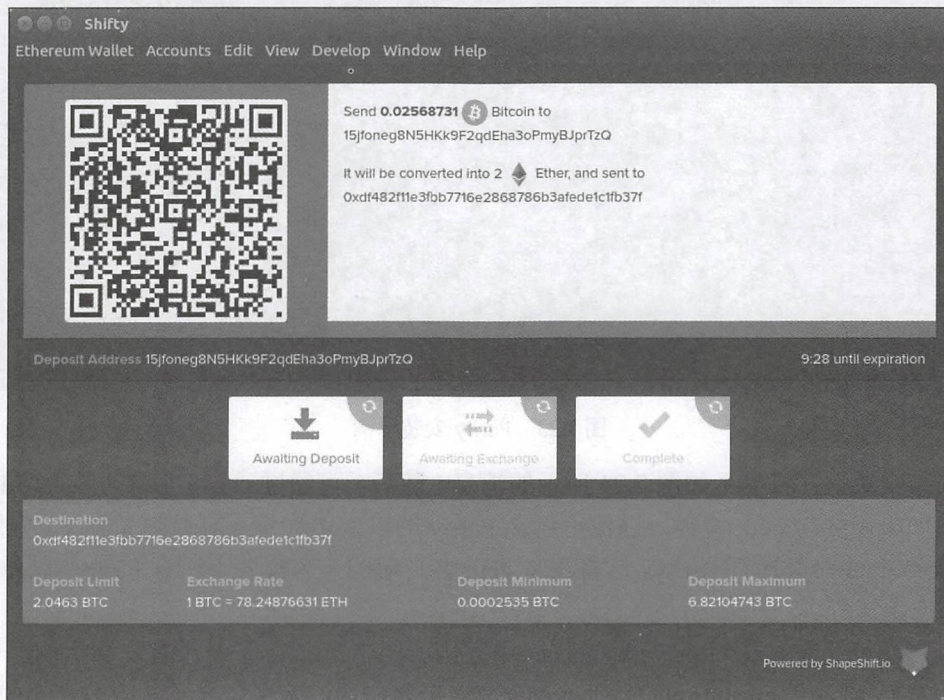


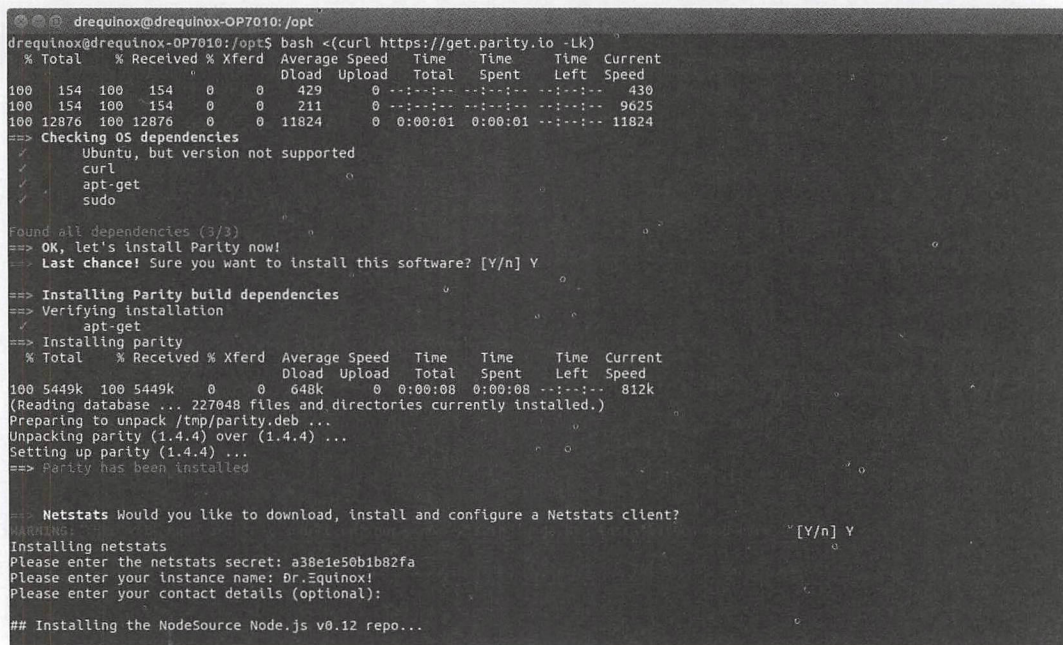
图 7.22 Ether 出现于当前账户中

- ❑ Parity 安装。Parity 是以太坊客户端的另一个实现，并采用 Rust 语言编写。Parity 的主要目的是高性能、小尺寸和可靠性。在 Ubuntu 或 Mac 系统上，可采用以下命令安装 Parity：

```
bash <(curl https://get.parity.io -Lk)
```

这将启动 Parity 客户端的下载和安装过程。Parity 安装完毕，安装程序还将提供 netstats 客户端的安装。netstat 客户端则是后台运行的守护进程，收集重要的统计信息并在 stats.ethdev.com 上对其予以显示。

图 7.23 显示了 Parity 安装示例。



```
drequinox@drequinox-OP7010: /opt$ bash <(curl https://get.parity.io -Lk)
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left   Speed
100 154    100 154    0    0    429      0  --:--:-- --:--:-- --:--:--  430
100 154    100 154    0    0    211      0  --:--:-- --:--:-- --:--:--  9625
100 12876  100 12876    0    0  11824      0  0:00:01  0:00:01 --:--:-- 11824
==> Checking OS dependencies
✓ Ubuntu, but version not supported
✓ curl
✓ apt-get
✓ sudo
Found all dependencies (3/3)
==> OK, let's install Parity now!
==> Last chance! Sure you want to install this software? [Y/n] Y
==> Installing Parity build dependencies
==> Verifying installation
✓ apt-get
==> Installing parity
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left   Speed
100 5449k  100 5449k    0    0  648k      0  0:00:08  0:00:08 --:--:--  812k
(Reading database ... 227048 files and directories currently installed.)
Preparing to unpack /tmp/parity.deb ...
Unpacking parity (1.4.4) over (1.4.4) ...
Setting up parity (1.4.4) ...
==> Parity has been installed

==> Netstats Would you like to download, install and configure a Netstats client?
WARNING:
Installing netstats
Please enter the netstats secret: a38e1e50b1b82fa
Please enter your instance name: Dr.Equinox!
Please enter your contact details (optional):

## Installing the NodeSource Node.js v0.12 repo...
```

图 7.23 Parity 安装示例

安装结束后，将显示如图 7.24 所示的消息。随后，可通过 `parity -j` 启动以太坊 Parity 节点。如果需要与 `geth` 兼容，以使用包含 Parity 的以太坊钱包（Mist 浏览器），则应使用 `parity -geth` 命令运行 Parity。这将在与 `geth` 客户端的兼容模式下运行 Parity，以使 Mist 在 Parity 上运行。

客户端可视需要在 <https://ethstats.net/> 中列出。图 7.25 显示了一个客户端示例。

图 7.26 显示了 ethstats.net 上列出的全部连接客户端，其中包含了对应客户端的相关属性，例如节点名称、节点类型、延迟、挖掘状态、对等点数量、未完成交易的数量、

最后一个区块。难度、区块交易以及 Uncles 的数量。

```

drequinox@drequinox-OP7010:/opt
[PM2] Spawning PM2 daemon with pm2_home=/home/drequinox/.pm2
[PM2] PM2 Successfully daemonized
[PM2][WARN] Applications node-app not running, starting...
[PM2] App [node-app] launched (1 instances)

  App name  id  mode  pid  status  restart  uptime  cpu  mem  watching
  node-app  0   fork  6018  online  0        0s      13%   18.2 MB  disabled

Use `pm2 show <id/name>` to get more details about an app

==> All done
==> Next steps
==> Run `parity -j` to start the Parity Ethereum client.

drequinox@drequinox-OP7010:/opt$

```

图 7.24 Parity 安装结束



图 7.25 <https://ethstats.net/> 中列出的某个客户端示例



图 7.26 <https://ethstats.net/> 中列出的全部连接客户端

Parity 还提供了一个用户友好的 Web 界面，并对不同的任务进行管理，例如账户管理、地址簿管理、DAPP 管理、合约管理、状态和签名者操作。

下列命令提供了 Parity 的访问功能。

```
$ parity ui
```

对应界面如图 7.27 所示。

如果在 geth 兼容模式下运行 Parity，则 Parity 的 UI 将处于禁用状态。为了使 UI 与 geth 兼容，可以使用以下命令：

```
$ parity --geth --force-ui
```


上述命令将在 geth 兼容模式下启动 Parity，另外还可启用 Web 用户界面。

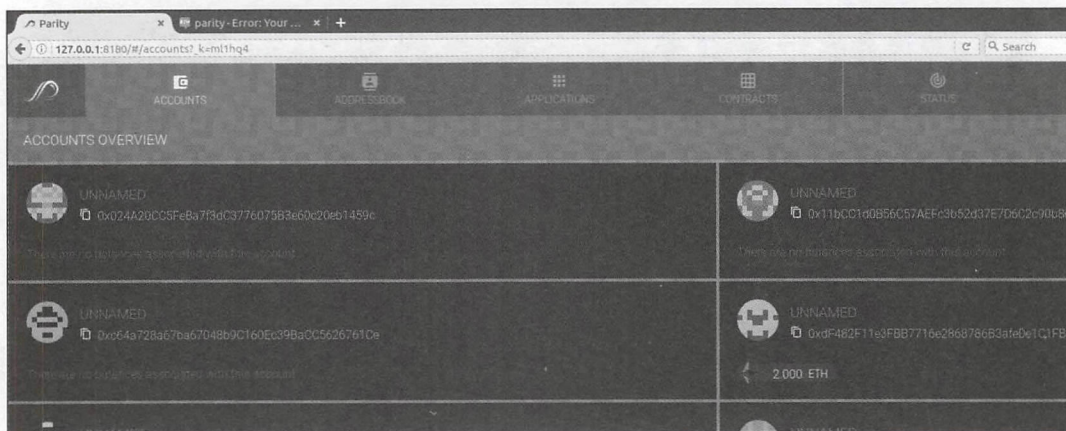


图 7.27 Parity 用户界面

利用 parity 命令行创建账户。下列代码利用 parity 创建了新的账户：

```
$ parity account new
Please note that password is NOT RECOVERABLE.
Type password:
Repeat password:
2016-11-30 02:18:55 UTC c8c92a910cfbce2e655c88d37a89b6657d1498fb
```

7.11 贸易与投资

Ether 可以在不同的交易平台购买和销售，在本书编写时，以太坊的市值为 680277967 英镑，Ether 的价格为 7.89 英镑。近期，由于以太坊攻击，以及以太坊网络上的分支（分叉），其价格变得非常不稳定且呈下降趋势。

图 7.28 显示了 Ether 的市值变化趋势。

Ether 可以在不同的交易平台购买，或对其进行开采。另外，某些在线服务还支持不同货币之间的兑换业务，例如 shapeshift.io。

与此同时，多家在线平台均提供了以法定货币购买的 Ether（如 kraken, Coinbase 等），并支持信用卡和其他虚拟货币业务，例如比特币。

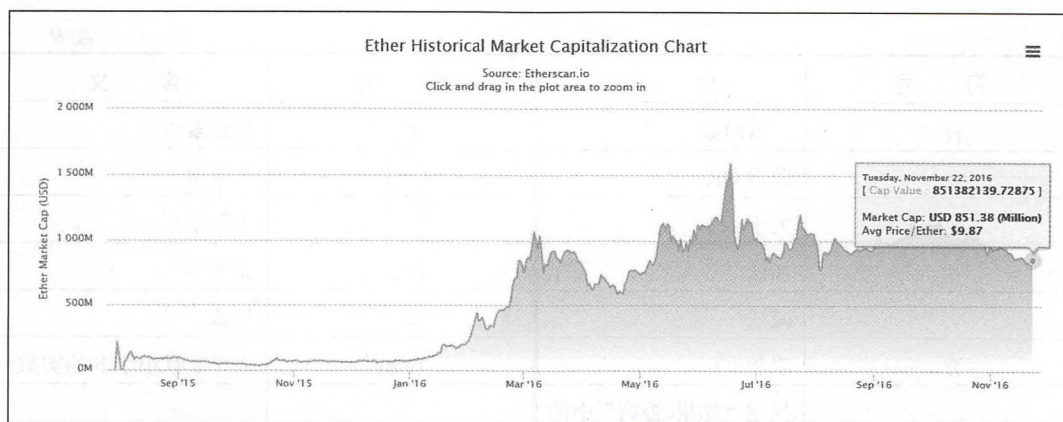


图 7.28 Ether 的市值（数据源自 Etherscan.io）

7.12 黄皮书

以太坊黄皮书由 Gavin Wood 博士编写，并作为以太坊协议的正式定义。若遵循黄皮书中的各项协议规范，任何人均可实现以太坊客户端。然而，相关内容稍显晦涩，对于缺少代数、数学、数学符号等背景知识的读者尤其如此。

针对于此，表 7.15 显示了黄皮书中的全部数学符号及其含义，以使阅读过程变得相对轻松，进而理解其中的各种概念和规范。

表 7.15 数学符号

符 号	含 义	符 号	含 义
\equiv	定义为	\leq	小于等于
$=$	等于	σ	全局状态
\neq	不等于	μ	机器状态
$ \dots $	长度	γ	以太坊状态转换函数
\in	属于某个集合	Π	区块级别状态转换函数
\notin	不属于某个集合	\cdot	序列连接
\forall	针对全部	\exists	存在
\cup	联合	Λ	合约生成函数
\wedge	逻辑 AND	Δ	增量
$:$	因此	$\{\}$	集合

续表			
符 号	含 义	符 号	含 义
()	元组函数	[]	数组索引
∨	逻辑 OR	[...]	下取整
>	大于	⌈...⌉	上取整
+	加法	...	无字节
-	减法	⊕	异或
Σ	求和	(a,b)	大于 a 且小于 b 的实数
{	描述“如果-否则”中的各种情况	∅	空集或 null

7.13 以太坊网络

以太坊网络是一个节点参与的点对点网络，以维护区块链并有助于达成共识机制。根据具体需求和使用情况，可以将以太坊网络分为 3 类。

7.13.1 MainNet

MainNet 是当前的以太坊网络，其版本为 homestead。

7.13.2 TestNet

TestNet 也称作 Ropsten，表示为以太坊区块链的测试网络。在部署至区块链之前，区块链用于测试智能合约和 DAPP。而且，作为测试网络，TestNet 还支持各种试验和研究行为。

7.13.3 专用网络

顾名思义，专用网络表示为一类私有网络，并可通过生成创始区块而构建。这一情况常出现于分布式账本网络中，其中，一组专属实体启动自身的区块链，并将其用作授

权许可的区块链。

关于专用网络的连接、测试以及配置，第8章将对此加以详细讨论。

7.14 所支持的协议

为了支持完整的去中心化生态系统，各种支持协议均处于完善中，其中包括 Whisper 和 Swarm 协议。除了合约层之外（这也是核心区块链层），还存在其他层须实现去中心化处理，进而形成完整的去中心化生态系统。其中涉及去中心化的存储机制和去中心化的消息机制。针对以太坊而开发的 Whisper 则是一项去中心化的消息协议；而 Swarm 则是一种去中心化的存储协议，旨在面向完整的去中心化 Web 提供基础内容。

1. Whisper 协议

Whisper 协议以太坊网络提供了去中心化的 P2P 消息传递功能。本质上，Whisper 协议是一种节点使用的通信协议，以实现彼此间的相互通信。另外，消息的数据和路由操作在 Whisper 通信中被加密。此外，Whisper 协议的设计理念面向小型数据传输，且无须实现实时通信。同时，Whisper 协议旨在提供一个无法追踪的通信层，并在双方之间提供“暗通信”。当然，区块链也可以用于通信，但代价昂贵，且共识机制并不需要在节点间交换消息。

当前，Whisper 可与 geth 协同使用。当运行 geth 以太坊客户端时，可通过--shh 选项启用 Whisper 协议。

2. Swarm 协议

Swarm 协议定义为一个分布式文件存储平台，是一个去中心化的、分布式 P2P 存储网络。该网络中的文件通过其内容的哈希结果进行处理。这与传统的集中式服务形成了鲜明的对比，后者仅在中心位置提供存储。同时，针对以太坊 Web 3.0 堆栈，这也构成了本地基础层服务。Swarm 协议实现了与 DevP2P 之间的集成，后者是以太坊的多协议网络层。Swarm 最初被设想为：向以太坊 Web 3.0 提供抗 DDOS（分布式拒绝服务）和容错分布式存储层。目前，Whisper 和 Swarm 协议仅处于开发状态中。尽管已针对 Swarm 协议发布了概念证明和 Alpha 版代码，但目前尚不存在稳定的版本。

图 7.29 显示了 Swarm 和 Whisper 协议的适配方式以及协同方式。

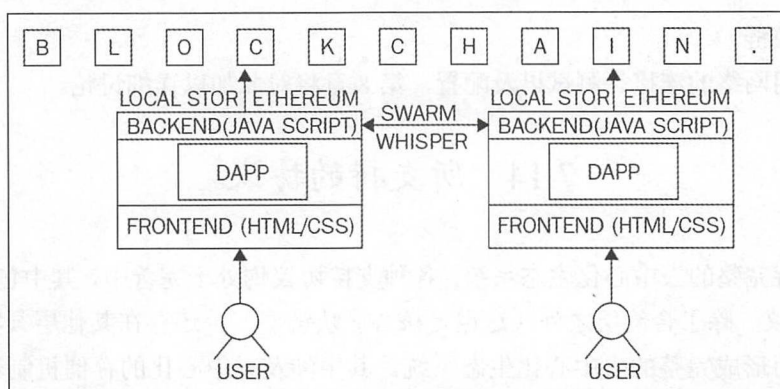


图 7.29 区块链、Whisper 和 Swarm 协议

7.15 以太坊应用程序

以太坊中存在各种各样的 DAO 和智能合约的实现方案，其中较有名气的则是 DAO。近期，DAO 被黑客入侵，并需要创建一个硬分叉恢复资金。该 DAO 可视为一个去中心化的平台，用于收集和分配投资。

Augur 则是另一个在以太坊中实现的 DAPP，同时也是一个去中心化的预测市场。关于其他去中心化应用程序，读者可访问 <http://dapps.ethercasts.com/> 以获取更多信息。

7.16 可扩展性和安全问题

区块链中的可扩展性都可视为一个基本问题；另外，安全问题同样不可小视；而隐私和保密等问题会引发某些适应性问题，尤其是在金融领域。与此同时，这些领域也投入了大量的研究工作。第 12 章还将对此加以详细讨论。

7.17 本章小结

本章首先讨论了以太坊的历史、研发动机，以及以太坊客户端；随后介绍了以太坊区块链的核心概念，如状态机模型、全局状态和机器状态、账户和账户类型。此外，还阐述了以太坊虚拟机（EVM）的核心组件，包括区块、区块结构、消息机制等概念。本

章后续部分考察了以太坊客户端的安装过程和管理方法，并引入了两种较为流行的客户端，即 `geth` 和 `Parity`。关于这一类客户端的进一步开发（基于以太坊），读者可参考第8章。最后，本章还讨论了支持协议，以及以太坊所面临的种种挑战。目前，以太坊仍处于持续发展中，这也离不开专业开发者社区的大力支持。关于以太坊最新的改进建议，读者可访问 <https://github.com/ethereum/EIPs>。此外，最近发起的企业以太坊联盟（EAA）旨在开发能够满足企业级业务需求的以太坊平台。随着对可伸缩性、优化、吞吐量、容量和安全性等问题的不断深入，可以相信，随着时间的推移，以太坊将演化为一个更加健壮、友好、稳定的区块链生态系统。

第 8 章 以太坊开发

本章将介绍与以太坊开发相关的概念、技术和工具。另外，相关示例将具体解释前述章节中的理论概念。同时，本章还将进一步考察开发环境的设置，以及如何使用以太坊区块链创建智能合约，并展示大量的操作实例，以帮助读者了解如何使用以太坊和其他支持工具，进而可在区块链中开发和部署智能契约。

8.1 配置开发环境

本章第一项任务是配置开发环境，随后介绍基于 TestNet（测试网络）和 PrivateNet（专用网络）的以太坊设置。这里，TestNet 也称作 Ropsten，并作为测试平台以供开发人员或用户使用，进而测试智能合约和其他与区块链相关的方案。以太坊专用网络选项允许创建一个独立的私有网络，用作参与实体之间的分布式账本，并用于开发和测试智能合约。需要说明的是，虽然存在其他可用于以太坊的客户端，例如 Parity，在第 7 章中曾有所提及，geth 是以太坊中的主要客户端，也是被选取的常用工具，因此本章将在示例中使用 geth。

8.1.1 TestNet (Ropsten)

以太坊 Go 客户端 geth 可通过下列命令连接至测试网络：

```
$ geth --TestNet
```

图 8.1 显示了示例输出结果，包含所选择的网络类型，以及关于区块链下载的其他信息。

```
lnran@reguinox-OP7010:~$ geth --testnet
I1204 16:03:32.759308 cmd/utils/flags.go:613] WARNING: No etherbase set and no accounts found as default
I1204 16:03:32.759415 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/lnran/.ethereum/testnet/geth/chaindata
I1204 16:03:32.807292 ethdb/database.go:176] closed db: /home/lnran/.ethereum/testnet/geth/chaindata
I1204 16:03:32.807589 node/node.go:175] instances: geth/v1.5.2-stable-c8695289/linux/go1.7.3
I1204 16:03:32.807693 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/lnran/.ethereum/testnet/geth/chaindata
I1204 16:03:32.814016 eth/backend.go:280] Successfully wrote custom genesis block: 0cd786a2425d16f152c658316c423e6ce1181e15c3295826d7c99
04c8a9ce303
I1204 16:03:32.814076 eth/db/upgrade.go:346] upgrading db log bloom bins
I1204 16:03:32.814112 eth/db/upgrade.go:354] upgrade completed in 36.513µs
I1204 16:03:32.814128 eth/backend.go:193] Protocol Versions: [63 62], Network Id: 2
I1204 16:03:32.914363 core/blockchain.go:214] Last header: #0 [0cd786a2...] TD=131072
I1204 16:03:32.814375 core/blockchain.go:215] Last block: #0 [0cd786a2...] TD=131072
I1204 16:03:32.814382 core/blockchain.go:216] Fast block: #0 [0cd786a2...] TD=131072
I1204 16:03:32.814840 p2p/server.go:336] Starting Server
I1204 16:03:37.983847 p2p/discover/udp.go:217] Listening, enode://fa838ec3fee8a26d75755b5f7cbdd80efacc4a98b5291acd5a23ae5465b794c84aff
e7be633524d2895768a2122a25e87cf97bd369895ace9f48f868eaeef18@[:]:30303
I1204 16:03:37.983960 p2p/server.go:604] Listening on [::]:30303
I1204 16:03:37.984963 node/node.go:340] IPC endpoint opened: /home/lnran/.ethereum/testnet/geth.ipc
I1204 16:04:17.984169 eth/downloader/downloader.go:326] Block synchronisation started
```

图 8.1 连接至测试网络的 geth 命令及其输入结果

8.1.2 配置 PrivateNet

专用网络将创建一个全新的区块链，这与 TestNet 或 MainNet 有所不同——使用创始区块和 Network ID。当创建专用网络时，需要使用到 3 个组件：

- ❑ Network ID。
- ❑ 创始文件。
- ❑ 存储区块链数据的目录。虽然并未对数据目录作强制要求，但若系统中已经存在多个区块链，则应该指定数据目录，以便新的区块链可使用一个单独的目录。

PrivateNet 可生成全新的区块链，这与 TestNet 和 MainNet 有所不同：后者使用自身的唯一创始区块以及 Network ID。在 MainNet 上，geth 在默认状态下知晓其他对等成员并自动链接；但在 PrivateNet 上，geth 需要制定相应的标记和配置内容，进而可被其他成员发现，或发现其他对等成员。

除了上述 3 个组件之外，还需禁用“节点发现”功能。据此，互联网上的其他节点将无法发现用户的私有网络，从而实现真正意义上的专用网络。如果其他网络恰好具有相同的创始文件和 Network ID，则可连接到用户的私人网络中。当然，这种概率非常低，但禁用“节点发现”功能是一项很好的习惯且被推荐使用。

下面将通过示例考察相关参数。

1. Network ID

Network ID 可以是任何正数，除了 1 和 3——已分别用于以太坊 MainNet 和 TestNet (Ropsten) 中。相应地，在本节后面讨论的示例专用网络中选择了 Network ID 786。

2. 创始文件

创始文件包含了自定义创始区块所需的必要字段。同时，这也是网络中的第一个区块，且不包含任何前区块。以太坊协议执行严格的检查，以确保互联网上不存在其他节点能够参与共识机制，除非它们具有相同的创始区块。

自定义创始区块如下所示：

```
{
  "nonce": "0x0000000000000042",
  "timestamp": "0x0",
  "parentHash": "0x000000000000000000000000000000000000000000000000",
  "extraData": "0x0",
  "gasLimit": "0x4c4b40",
  "difficulty": "0x400",
```



```

"mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
"coinbase": "0x0000000000000000000000000000000000000000000000000000000000000000",
"alloc": { }
}

```

该文件可以保存在具有 JSON 扩展名的文本文件中，例如 `privategenesis.json`。作为可选方案，可通过指定受益人地址和 `alloc` 中的 Wei 数量分配 Ether。但通常并无此类必要，因为在专用网络上，Ether 可以实现快速开采。

3. 数据目录

该目录用于保存私有以太坊网络中的区块链数据。例如，在后续示例中，对应目录定义为 `~/ethereum/privatenet`。

geth 客户机指定了一些参数，以便启动、进一步调整配置内容，并启用私有网络。相关标志如下所示。

- ❑ `--nodiscover`: 如果节点恰好具有相同的创始文件和 Network ID，该标志可以确保节点不会被自动发现。
- ❑ `--maxpeers`: 此标记用于指定允许连接到私有网络的对等点数量。如果设置为 0，则其他人无法建立连接。这适用于某些场合，例如私有测试。
- ❑ `--rpc`: 用于启用 geth 中的 RPC 界面。
- ❑ `--rpcapi`: 该标记接收 API 列表作为参数。例如，`eth、web3` 将在 RPC 上启用 `web3` 和 `3th` 界面。
- ❑ `--rpcport`: 用于设置 TCP RPC 端口，例如 9999。
- ❑ `--rpccorsdomain`: 该标记用于指定 URL，进而连接至私有 geth 节点，并执行 RPC 操作。
- ❑ `--port`: 该标记用于指定 TCP 端口，进而监听源自其他对等点成员的接入连接。
- ❑ `--identity`: 该标记定义为一个字符串，以指定私有节点的名称。

4. 静态节点

如果打算连接一组特定的对等点，此类节点可添加至一个文件中，并于其中保存区块链数据和密钥存储文件，例如 `~/ethereum/privatenet` 目录。此处，文件名应确定为 `static-nodes.json`，这在私有网络中十分有用。下列内容显示了 json 文件示例。

```

[
  "enode://
44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233
c8d426fc57a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f@xxx
.xxx.xxx:TCP_PORT"
]

```

其中, xxx 表示为公共 IP 地址, TCP_PORT 可以是系统上的任何有效和可用的 TCP 端口。另外, 长十六进制字符串表示节点 ID。

8.1.3 启动私有网络

下列内容显示了启动私有网络的相关命令。

```
$ geth --datadir ~/.ethereum/privatenet init
./privether/privategenesis.json
```

图 8.2 显示了输出结果。

```
imran@drequinox-OP7010:~$ geth --datadir ~/.ethereum/privatenet init ./privether/privategenesis.json
I0211 23:49:15.319145 cmd/utlils/flags.go:613] WARNING: No etherbase set and no accounts found as default
I0211 23:49:15.319582 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/.ethereum
/privatenet/geth/chaindata
I0211 23:49:15.350823 ethdb/database.go:176] closed db:/home/imran/.ethereum/privatenet/geth/chaindata
I0211 23:49:15.350862 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/.ethereum
/privatenet/geth/chaindata
I0211 23:49:15.357049 cmd/geth/main.go:256] successfully wrote genesis block and/or chain rule set: f2b2ffed019
07a845a01d1dea21e5a81360ae3c38ec021e8e68b5ec9ffcc82df
imran@drequinox-OP7010:~$
```

图 8.2 私有网络的初始化结果

对应输出结果表明, 创始区块已成功被创建。下列命令可用于启动 geth。

```
$ geth --datadir .ethereum/privatenet/ --networkid 786
```

对应结果如图 8.3 所示。

```
imran@drequinox-OP7010:~$ geth --datadir .ethereum/privatenet/ --networkid 786
I0211 23:52:02.322730 cmd/utlils/flags.go:613] WARNING: No etherbase set and no accounts found as default
I0211 23:52:02.322814 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/.ethereu
/privatenet/geth/chaindata
I0211 23:52:02.378525 ethdb/database.go:176] closed db:/home/imran/.ethereum/privatenet/geth/chaindata
I0211 23:52:02.378825 node/node.go:175] instance: Geth/v1.5.2-stable-c8695209/linux/go1.7.3
I0211 23:52:02.378838 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/.ethereu
/privatenet/geth/chaindata
I0211 23:52:02.388823 eth/db_upgrade.go:346] upgrading db log bloom bins
I0211 23:52:02.388906 eth/db_upgrade.go:354] upgrade completed in 85.497ps
I0211 23:52:02.388930 eth/backend.go:193] Protocol Versions: [63 62], Network Id: 786
I0211 23:52:02.389234 core/blockchain.go:214] Last header: #0 [f2b2ffed...] TD=512
I0211 23:52:02.389251 core/blockchain.go:215] Last block: #0 [f2b2ffed...] TD=512
I0211 23:52:02.389264 core/blockchain.go:216] Fast block: #0 [f2b2ffed...] TD=512
I0211 23:52:02.399750 p2p/server.go:336] Starting Server
I0211 23:52:07.553857 p2p/discover/udp.go:217] Listening, enode://9efe46a58bc3f3d7c8d2ab0e18244c9f72bccdee9300
9751810dea32b8cd8465a0ac3685ea48871ec797f9f534912b2d1b983da25993b27397ecc1abdd54678[:]:30303
I0211 23:52:07.554074 p2p/server.go:604] Listening on [::]:30303
I0211 23:52:07.555359 node/node.go:340] IPC endpoint opened: /home/imran/.ethereum/privatenet/geth.ipc
```

图 8.3 针对私有网络启动 geth

当前, geth 可以通过 IPC 连接到私有网络上运行的 geth 客户端。通过下面的命令, 即可与私有网络上运行的 geth 会话进行交互。

```
$ geth attach ipc:.ethereum/privatenet/geth.ipc
```


这将开启交互式 JavaScript 控制台，并运行 PrivateNet 会话，如图 8.4 所示。

```
amran@drequinox-OP7010:~$ geth attach ipc:.ethereum/privatenet/geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.5.2-stable-c8695209/linux/go1.7.3
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0
>
```

图 8.4 启动 geth 并连接私有网络 786

不难发现，当 geth 启动时，图中显示了一条警告消息。



警告：

默认状态下不存在 etherbase 和账户。

出现该消息的原因在于，目前新测试网络中不存在可用的账户，且不存在相关账户设置为 etherbase，以接收挖掘奖励。针对这一问题，可创建一个新账户并将其设置为 etherbase。

当在测试网络上进行挖掘时，同样需要执行类似的操作。注意，此类命令须在 geth JavaScript 控制台输入。

下列命令用于创建新账户，在当前环境下，该账户创建于 Private Network ID 786 上。

```
> personal.newAccount("Password123")
"0x76f11b383dbc3becf8c5d9309219878caae265c3"
```

当账户创建完毕后，下一个步骤是将其设置为 etherbase/coinbase 账户，从而接收挖掘奖励。对应命令如下：

```
> miner.setEtherbase(personal.listAccounts[0])
true
```

当前，etherbase 账户不包含任何余额，并可通过下列命令进行查看。

```
> eth.getBalance(eth.coinbase).toNumber();
0
```

最后，启动挖掘过程的相关命令接收一个参数，表示线程的数量。在下面的例子中，两个线程将分配与挖掘过程。也就是说，在 start 函数中指定 2 作为参数，如下所示。

```
> miner.start(2)
true
```

待挖掘启动后，将生成首个 DAG，如图 8.5 所示。

[illegible]

图 8.5 生成 DAG

待 DAG 生成完毕后，即启动挖掘过程，geth 将输出如图 8.6 所示的结果。不难发现，被成功挖掘的区块包含“Mined 5 blocks...”这一消息内容。

```
I1204 22:38:02.373804 miner/worker.go:436] ⚡ Mined 5 blocks back: block #487
I1204 22:38:02.373908 miner/worker.go:542] commit new work on block 493 with 0 txs & 0 uncles. Took 86.005µs
I1204 22:38:02.637297 miner/worker.go:344] ⚡ Mined block (#493 / 9a95245e). Wait 5 blocks for confirmation
I1204 22:38:02.637415 miner/worker.go:542] commit new work on block 494 with 0 txs & 0 uncles. Took 91.009µs
I1204 22:38:02.637436 miner/worker.go:436] ⚡ Mined 5 blocks back: block #488
I1204 22:38:02.639064 miner/worker.go:542] commit new work on block 494 with 0 txs & 0 uncles. Took 1.609044ms
I1204 22:38:03.538525 miner/worker.go:344] ⚡ Mined block (#494 / cb89cccd). Wait 5 blocks for confirmation
I1204 22:38:03.538719 miner/worker.go:542] commit new work on block 495 with 0 txs & 0 uncles. Took 158.751µs
I1204 22:38:03.538745 miner/worker.go:436] ⚡ Mined 5 blocks back: block #489
I1204 22:38:03.538860 miner/worker.go:542] commit new work on block 495 with 0 txs & 0 uncles. Took 95.822µs
I1204 22:38:03.548923 miner/worker.go:344] ⚡ Mined block (#495 / 53d98079). Wait 5 blocks for confirmation
I1204 22:38:03.549064 miner/worker.go:542] commit new work on block 496 with 0 txs & 0 uncles. Took 120.447µs
I1204 22:38:03.549082 miner/worker.go:436] ⚡ Mined 5 blocks back: block #490
I1204 22:38:03.549159 miner/worker.go:542] commit new work on block 496 with 0 txs & 0 uncles. Took 64.047µs
```

图 8.6 挖掘过程的输出结果

下列命令可终止挖掘操作。

```
> miner.stop
true
```

在 JavaScript 控制台中，可以查询全部 Ether 的余额。经挖掘后，可以查看到余额总量的显著变化。鉴于采用了私有网络和创始文件，因而挖掘速度非常快，且所设定的网络难度也很低，如下所示：

```
> eth.getBalance(eth.coinbase).toNumber();
2.72484375e+21
```

如果在序列中按下两次 Space 键和两次 Tab 键，就会显示一个完整的可用对象列表，如图 8.7 所示。

Array	Math	TypeError	constructor	hasOwnProperty	parseFloat	toString
BigNumber	NaN	URIError	debug	inspect	parseInt	txpool
Boolean	Number	Web3	decodeURI	isFinite	personal	undefined
Date	Object	setInterval	decodeURIComponent	isNaN	propertyIsEnumerable	unescape
Error	RangeError	setTimeout	encodeURI	isPrototypeOf	require	valueOf
EvalError	ReferenceError	admin	encodeURIComponent	join	rpc	web3
Function	RegExp	clearInterval	escape	loadScript	setInterval	
Infinity	String	clearTimeout	eth	miner	setTimeout	
JSON	SyntaxError	console	eval	net	toLocaleString	

图 8.7 可用对象

此外，当输入一条命令时，可按下 Tab 键两次自动完成该命令。如果按下两次 Tab


```

imran@drequinox-OP7010: /opt/Ethereum Wallet
imran@drequinox-OP7010:/opt/Ethereum Wallet$ ./Ethereum\ Wallet --rpc /home/imran/.ethereum/privatenet/geth.ipc
[2016-12-06 07:58:08.706] [INFO] main - Running in production mode: true
Secp256k1 bindings are not compiled. Pure JS implementation will be used.
[2016-12-06 07:58:08.866] [INFO] main - Starting in Wallet mode
[2016-12-06 07:58:08.932] [INFO] Db - Loading db: /home/imran/.config/Ethereum Wallet/mist.lokidb
[2016-12-06 07:58:08.947] [INFO] Windows - Creating commonly-used windows
[2016-12-06 07:58:09.948] [INFO] Windows - Create secondary window: loading, owner: notset
[2016-12-06 07:58:09.912] [INFO] updateChecker - Check for update...
[2016-12-06 07:58:11.373] [INFO] Windows - Create primary window: main, owner: notset
[2016-12-06 07:58:11.385] [INFO] Windows - Create primary window: splash, owner: notset
[2016-12-06 07:58:11.989] [INFO] ipcCommunicator - Backend language set to: en-GB
[2016-12-06 07:58:13.199] [INFO] (ui: splash) - Web3 already initialized, re-using provider.
[2016-12-06 07:58:13.362] [INFO] ClientBinaryManager - Initializing...
[2016-12-06 07:58:13.363] [INFO] ClientBinaryManager - Resolving path to Eth client binary ...
[2016-12-06 07:58:13.363] [INFO] ClientBinaryManager - Eth client binary path: /opt/Ethereum Wallet/nodes/eth/linux-x64/eth
[2016-12-06 07:58:13.663] [INFO] ClientBinaryManager - Initializing...
[2016-12-06 07:58:13.664] [INFO] ClientBinaryManager - Resolving platform...
[2016-12-06 07:58:13.664] [INFO] ClientBinaryManager - Calculating possible clients...
[2016-12-06 07:58:13.667] [INFO] ClientBinaryManager - 1 possible clients.
[2016-12-06 07:58:13.667] [INFO] ClientBinaryManager - Verifying status of all 1 possible clients...
[2016-12-06 07:58:13.669] [INFO] ClientBinaryManager - Verify Geth status ...
[2016-12-06 07:58:13.691] [INFO] ClientBinaryManager - Checking for Geth sanity check ...
[2016-12-06 07:58:13.693] [INFO] ClientBinaryManager - Checking sanity for Geth ...
[2016-12-06 07:58:13.764] [INFO] Sockets/node-ipc - Connect to {"path":"/home/imran/.ethereum/privatenet/geth.ipc"}
[2016-12-06 07:58:13.768] [INFO] Sockets/node-ipc - Connected!
[2016-12-06 07:58:13.769] [INFO] NodeSync - Ethereum node connected, re-start sync
[2016-12-06 07:58:13.776] [INFO] NodeSync - Starting sync loop
[2016-12-06 07:58:13.771] [INFO] Sockets/7 - Connect to {"path":"/home/imran/.ethereum/privatenet/geth.ipc"}
[2016-12-06 07:58:13.772] [INFO] main - Connected via IPC to node.
[2016-12-06 07:58:13.801] [INFO] Sockets/7 - Connected!
[2016-12-06 07:58:13.818] [INFO] (ui: splash) - network is privatenet
[2016-12-06 07:58:14.939] [INFO] updateChecker - App is up-to-date.

```

图 8.10 运行以太坊钱包并连接至 Private Net

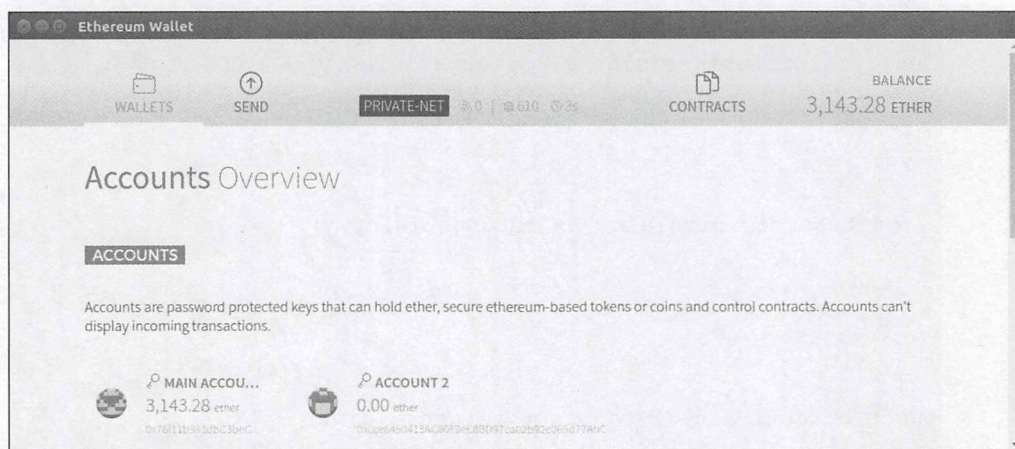


图 8.11 Private Net 上的 Mist

通过 RPC, Mist 也可运行在当前网络上。如果 geth 和 Mist 分别运行在不同的节点上,这一功能将十分有效。对此,可利用下列标记运行 Mist:

```
--rpc http://127.0.0.1:8545
```

8.1.5 利用 Mist 部署合约

使用 Mist 来部署新合约并不复杂。Mist 提供了一个界面,其中可以采用 Solidity 编

写合约，然后在网络上进行部署。

在当前示例中，可以使用一个较为简单的合约，在输入参数上执行各种简单的算术运算。下面讨论如何使用 Mist 部署合约及其相关步骤。由于目前阶段尚未引入 Solidity，因而当前仅是体验合约的部署和交互过程。后续内容将提供更多关于编码和 solidity 方面的信息，进而降低代码的理解难度。熟悉 JavaScript 或类似语言的读者将会发现，代码大多具有自解释特征。

下列内容显示了合约源代码示例。

```
pragma solidity ^0.4.0;
contract SimpleContract2
{
    uint x;
    uint z;
    function addition(uint x) returns (uint y)
    {
        z=x+5;
        y=z;
    }
    function difference(uint x) returns (uint y)
    {
        z=x-5;
        y=z;
    }
    function division(uint x) returns (uint y)
    {
        z=x/5;
        y=z;
    }

    function currValue() constant returns (uint)
    {
        return z;
    }
}
```

上述代码可简单地复制至合约部分下方的 Mist 中，如图 8.12 所示。在图 8.12 中左侧，可于其中复制代码。经检查且无语法错误后，合约部署选项将出现于右侧下拉菜单中，即“SELECT CONTRACT TO DEPLOY”。随后，简单地选择当前合约，并单击屏幕下方的 Deploy 按钮即可。

随后，Mist 将询问账户密码，并显示如图 8.13 所示的窗口。

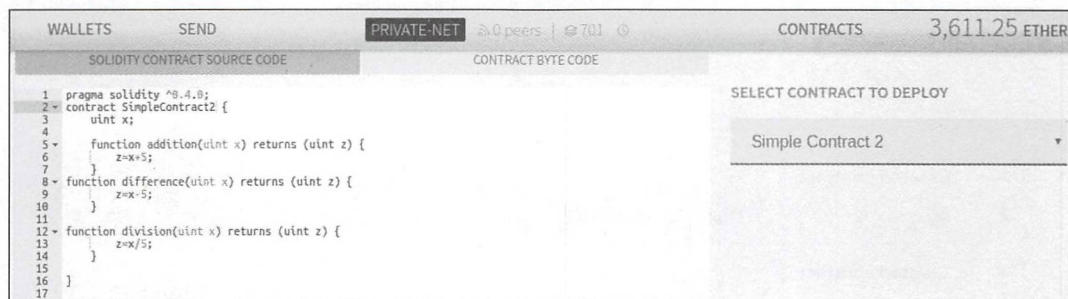


图 8.12 Mist 浏览器中的合约部署

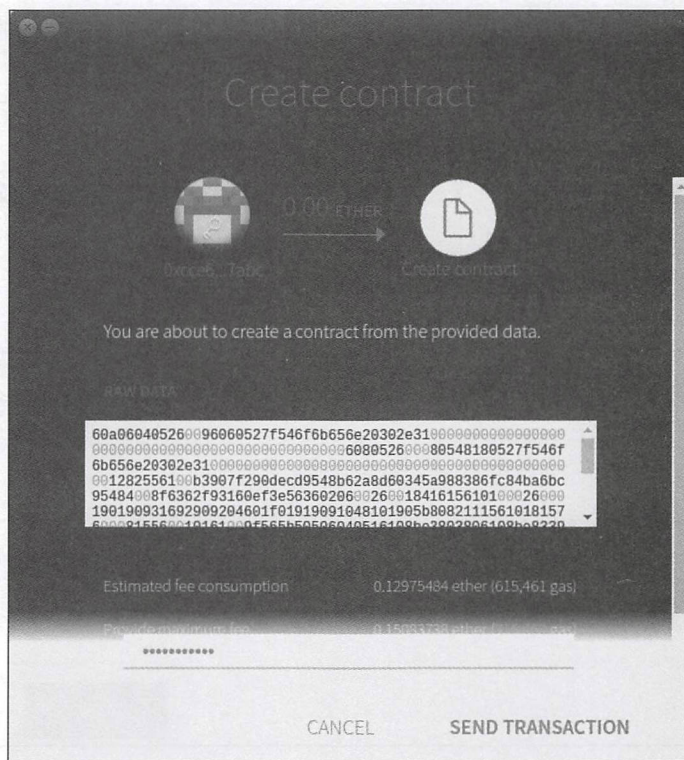


图 8.13 利用 Mist 创建合约

输入密码并单击 SEND TRANSACTION 部署当前合约。

部署及挖掘成功后，将在 Mist 中显示相应的交易列表，如图 8.14 所示。

待合约生成完毕后，即可通过执行交易，以及调用相关函数（基于 Mist）进行交互，如图 8.15 所示。

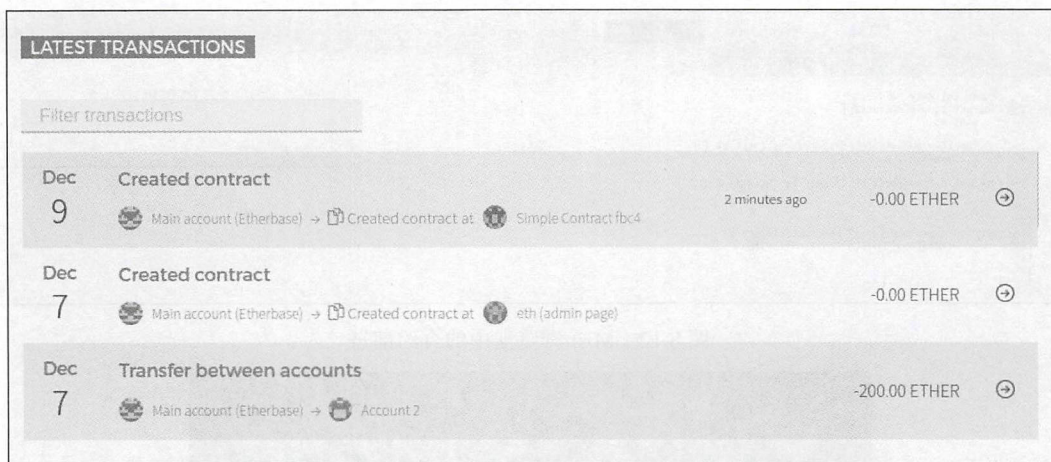


图 8.14 Mist 中的交易列表

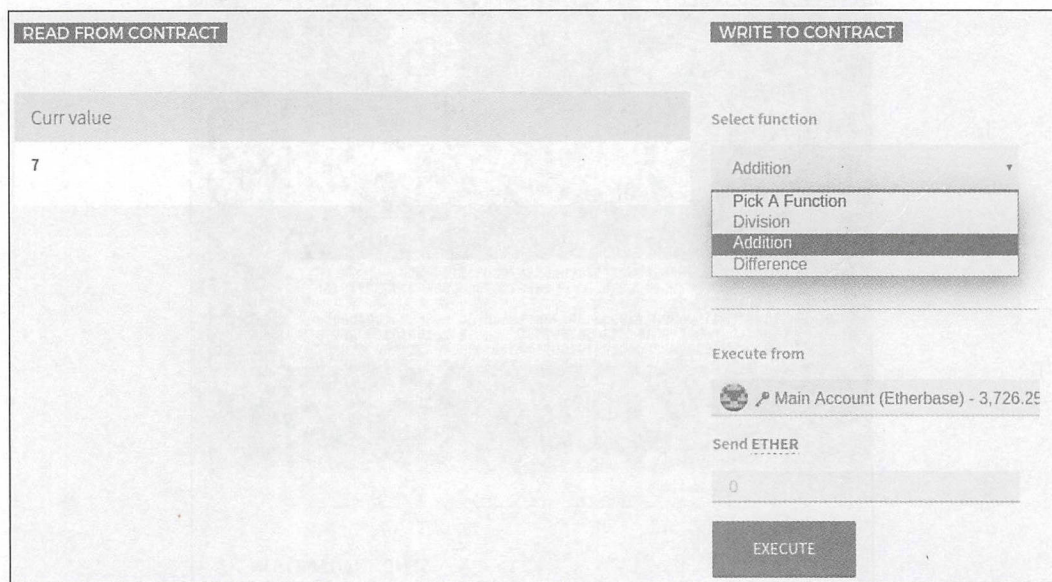
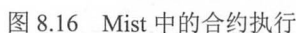


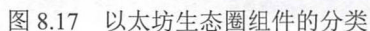
图 8.15 通过 Mist 中的读、写选项与合约进行交互

图 8.15 中, READ FROM CONTRACT 和 WRITE TO CONTRACT 选项处于可用状态。除此之外,图 8.15 中右侧还显示了当前合约所包含的各项功能。当所需函数选取完毕后,即可针对其输入相应值;随后可选取当前账户(据此开始执行)。接下来,单击 EXECUTE 按钮即可执行当前交易,这将调用合约中所选的函数。

上述过程的处理结果如图 8.16 所示。



以太坊开发涉及多种工具，图 8.17 显示了以太坊的各种开发工具、客户端、IDE 和开发框架的分类。



在本章中，主要关注的是 geth、浏览器的 Solidity、Solidity、solc 和 truffle。其他内容仅做简要介绍。

8.2.1 开发语言

合约可通过多种语言实现，读者可考虑采用下列 4 种语言之一编写合约。

- ❑ Mutan: 一种 G 风格的编程语言，于 2015 年被弃用且目前不再使用。
- ❑ LLL: 一种 Lisp 风格的编程语言，这也是其名称的由来，目前也不再使用。
- ❑ Serpent: 一种简洁的类 Python 语言，在合约开发方面应用较为活跃。
- ❑ Solidity: 这种语言现已成为编写以太坊合同的标准，也是本章的重点内容。

8.2.2 编译器

编译器可将高级合约源代码转换为以太坊执行环境所理解的格式。其中，Solidity 编译器是一类最常见的编译器，下面将对其予以分析。

1. Solc

Solidity 编译器将 Solidity 高级语言转换为以太坊虚拟机（EVM）字节码，以便可以通过 EVM 在区块链上执行。

Linux Ubuntu 操作系统上的 Solidity 编译器可以使用以下命令来安装：

```
$ sudo apt-get install solc
```

若 PPA 尚未安装，则可运行下列命令执行安装：

```
sudo add-apt-repository ppa:ethereum/ethereum  
sudo apt-get update
```

当检测 Solidity 编译器的现有版本及其安装状态时，可执行下列命令：

```
$ solc --version  
solc, the solidity compiler commandline interface  
Version: 0.4.6+commit.2dabddf0.Linux.g++
```

Solc 支持多项功能，例如：

- ❑ 以二进制格式显示合约，如图 8.18 所示。
- ❑ 估算 gas 量，如下所示：

```
imran@drequinox-OP7010:~$ solc --gas contract1.sol  
===== SimpleContract =====  
Gas estimation:  
construction:  
97 + 54600 = 54697
```


图 8.19 中左侧为包含语法高亮和代码格式化功能的代码编辑器；图 8.19 中右侧显示了各类可用的工具，可以用来部署、调试、测试并与合约进行交互，其中涵盖了各种特性，例如交易交互、JavaScript VM 连接选项、执行环境的配置、调试器、形式验证和静态分析。随后，经配置后，可连接至诸如 JavaScript VM 等执行环境。Web3 中的 Mist 或类似环境均提供了相应的执行环境，或者 Web3 提供方，进而可通过 HTTP 上的 IPC 或 RPC（Web3 提供方端点）连接至本地运行的以太坊客户端（例如 geth）。

- ❑ **Remix.** Remix IDE 在 2016 年 8 月停止后，目前该项目又重新启动。Remix 是目前正在开发中的、基于浏览器的 IDE，仅其调试器部分可用。Remix 调试器的功能非常强大，可用于对 EVM 字节码执行细节级别的跟踪和分析。后续章节还将介绍 Remix 安装和应用示例。
- ❑ **安装过程。**读者可访问 <https://github.com/ethereum/remix> 下载 Remix。随后第一步是复制 GitHub 存储库，如下所示：

```
$ git clone https://github.com/ethereum/remix
Cloning into 'remix'...
remote: Counting objects: 2185, done.
remote: Compressing objects: 100% (213/213), done.
remote: Total 2185 (delta 124), reused 0 (delta 0), pack-reused 1971
Receiving objects: 100% (2185/2185), 1.12 MiB | 443.00 KiB/s, done.
Resolving deltas: 100% (1438/1438), done.
Checking connectivity... done.
```

执行成功后，即可运行下列命令：

```
cd remix
npm install
npm run build
```

此时，可运行 `npm run start_node`，或者相关标记启动 geth。一旦 geth 启动并运行，就可以运行一个简单的 Web 服务器，以服务于 Remix Web 页面。

通过下列命令可启动 geth：

```
$ geth --datadir .ethereum/privatenet/ --networkid 786 --rpc --rpcapi
'web3,eth,debug' --rpcport 8001 --rpccorsdomain 'http://localhost:7777'
```

此处应留意--rpcapi 标记，该标记支持 RPC 上的 web3、eth 以及 debug。

当运行 `npm run start_node` 时，将显示下列消息。

```
$ npm run start_node
> ethereum-remix@0.0.2-alpha.0.0.9 start_node /home/imran/remix
> ./runNode.sh
both eth and geth has been found in your system
```



```
restart the command with the desired client:  
npm run start_eth  
or  
npm run start_geth
```

假设此处需要使用到 geth，则可输入下列命令：

```
$ npm run start_geth
```

如果选择运行 geth，则需要一个简单的 Web 服务器浏览 Remix Web 页面。对此，只需发布一条 Python 命令即可，并从 Remix 目录中运行，如图 8.20 所示。

```
imran@drequinox-0P7010:~/remix$ python -m SimpleHTTPServer 7777  
Serving HTTP on 0.0.0.0 port 7777 ...
```

图 8.20 Python Web 服务器

一旦上述命令和 Web 服务器成功运行，Remix 可通过 `http://localhost:7777` URL 进行浏览，如图 8.21 所示。

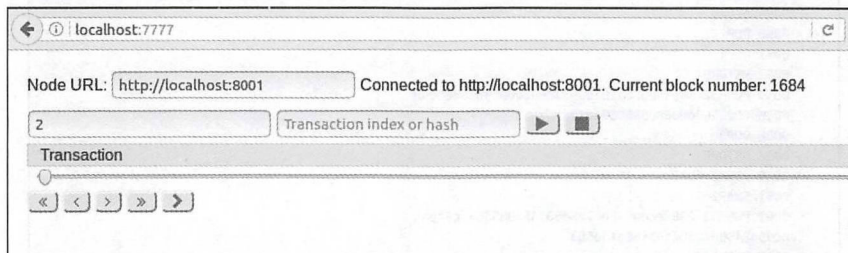


图 8.21 Web 浏览器显示 Remix 运行状态，并通过 TCP 7777 提供服务

另外，Remix 也是浏览器 Solidity 中的部分内容（关于浏览器 Solidity，读者可参考前述章节），并可通过提供 web3 提供方端点连接至本地 PrivateNet，如图 8.22 所示。

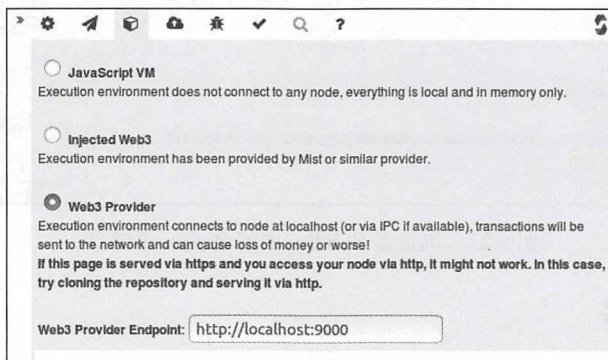


图 8.22 Web3 提供方

1. Node.js version 7

大多数工具和库都需要使用到 Node.js，可通过以下命令进行安装：

```
curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

2. 本地以太坊区块浏览器

本地以太坊区块资源管理器是一个有用的工具，可以用来浏览本地的区块链。可以按照以下各项步骤进行安装。

在 Linux Ubuntu 设备上，运行以下命令安装本地以太坊区块资源管理器。

```
$ git clone https://github.com/etherparty/explorer
```

这将显示如下输出结果：

```
Cloning into 'explorer'...  
remote: Counting objects: 253, done.  
remote: Total 253 (delta 0), reused 0 (delta 0), pack-reused 253  
Receiving objects: 100% (253/253), 51.20 KiB | 0 bytes/s, done.  
Resolving deltas: 100% (130/130), done.  
Checking connectivity... done.
```

下一步是将路径更改为 explorer 并运行以下命令：

```
imran@drequinox-OP7010:~$ cd explorer/  
imran@drequinox-OP7010:~/explorer$ npm start  
> EthereumExplorer@0.1.0 prestart /home/imran/explorer  
> npm install
```

待安装结束后，将显示如图 8.24 所示的结果，并启动基于以太坊浏览器的 HTTP 服务器。

```
> EthereumExplorer@0.1.0 start /home/imran/explorer  
> http-server ./app -a localhost -p 8000 -c-1  
  
Starting up http-server, serving ./app on port: 8000  
Hit CTRL-C to stop the server
```

图 8.24 以太坊浏览器的 HTTP 服务器

当 Web 服务器处于运行状态，可通过下列命令启动 geth。

```
geth --datadir .ethereum/privatenet/ --networkid 786 --rpc  
--rpccorsdomain 'http://localhost:8000'
```

在成功启动 `geth` 后，可定位至 TCP 端口 8000 上的本地服务器，以访问本地以太坊区块浏览器，如图 8.25 所示。

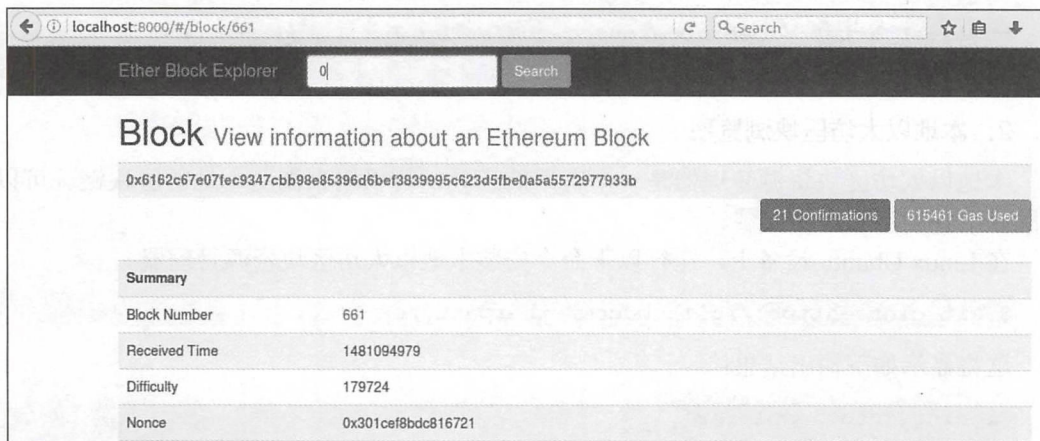


图 8.25 区块浏览器

除此之外，还可采用 Python 等启动 Web 服务器。在 Python 中，可通过下列命令启动 Web 服务器。

```
imran@drequinox-OP7010:~/explorer/app$ python -m SimpleHTTPServer 9900
Serving HTTP on 0.0.0.0 port 9900 ...
```

同时，启动 `geth` 客户端须使用到相关参数，否则，将会出现如图 8.26 所示的错误消息。

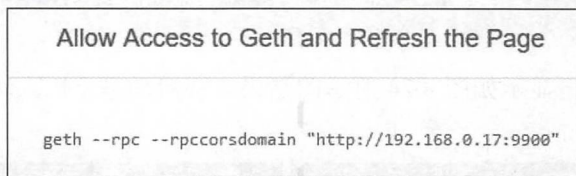


图 8.26 以太坊本地区块浏览器中的错误消息

重启 `geth` 以支持 `rpccorsdomain`，如下所示：

```
geth --datadir .ethereum/PrivateNet/ --networkid 786 --rpc --rpccorsdomain
'http://192.168.0.17:9900'
```

8.2.4 EthereumJS

在某些场合下，在 TestNet 和 MainNet 上进行测试，显然不是一个测试合约的理想位

置。专用网络有时会耗费大量的时间，当需要快速测试且缺少合适的测试网络时，EthereumJS testrpc 可以派上用场，并采用 EthereumJS 模拟以太坊 geth 客户端的行为，同时支持更快的开发测试。testrpc 可以通过 npm 作为一个节点包提供。

在安装 testrpc 之前，应先期安装 Node.js，并且 npm 包管理器也应该可用。

testrpc 可通过下列命令进行安装：

```
npm install -g ethereumjs-testrpc
```

为了启动 testrpc，只需输入该命令并保持其在后台运行，并打开另一个终端以处理合约。

```
$testrpc
```

8.2.5 合约的开发和部署

开发和部署合约需要采取多项步骤。一般来讲，该过程可以分为 4 个步骤：编写、测试、验证和部署。在部署技术之后，下一个步骤是创建用户界面，并通过 Web 服务器将其呈现给最终用户。

其中，编写步骤涉及在 Solidity 中编写合约源代码，并可以在任何文本编辑器中完成。Linux、Atom 和其他编辑器均提供了各种插件和附加组件，并为 Solidity 源代码提供了语法高亮和格式化等功能。

测试通常采用自动化的方式进行。后续章节将会讨论 truffle，并使用 Mocha 框架测试合约。除此之外，也可以采用手工方式进行测试。一旦合同在模拟环境（如 EthereumJS testrpc）或 PrivateNet 中被验证、执行和测试，随后即可部署至 Ropsten 测试网以及最终的区块链（Homestead）上。

在 8.3 节中，将介绍 Solidity 语言，并提供了编写合约所需的基础知识。Solidity 语言的语法类似于 C 和 JavaScript，且易于进行编程。

8.3 Solidity 语言

Solidity 是一种特定于某一领域的编程语言，用于以太坊合约编程。除此之外，其他语言还包括 serpent、Mutant 和 LLL。在本书编写时，Solidity 语言则更为常见。其语法更接近于 JavaScript 和 C 语言。在过去的几年里，Solidity 已经发展成为一种成熟的语言，且易于使用。尽管如此，Solidity 仍然有很长的路要走，例如，某些高级特性和功能方面的完善。但就合约编程而言，Solidity 依然是目前最广泛使用的编程语言。

Solidity 是一种静态类型化语言，这意味着在编译时，可以在 Solidity 中进行变量类型检查。每个变量都必须在编译时用一个类型来指定。因此，验证和检查过程将在编译时完成；某些特定的 bug，如数据类型的解释，可于先期被发现，而非运行期。后者的代价较为高昂，对于区块链/智能合约范例尤其如此。Solidity 语言的其他特性还包括继承、库和定义复合数据类型的能力。

Solidity 是一种面向合约的语言。在 Solidity 中，合约等价于其他面向对象编程语言中的类。

Solidity 包含两种数据类型分类，即值类型和引用类型。

8.3.1 值类型

本小节将对各种值类型记忆分析。

1. 布尔类型

布尔类型包含两种可能值，即 true 或 false。例如：

```
bool v = true;
```

该语句将 true 赋予 v。

2. 整型值

该数据类型表示为整数，表 8.1 显示了声明整数数据类型时所用的各种关键字。

表 8.1 声明整数类型

关 键 字	类 型	描 述
int	有符号整数	int8~int256，这意味着关键字可以从 int8 增至 int256（以 8 为单位增加）。例如，int8、int16、int24
uint	无符号整数	uint8~uint256

例如，在下列示例代码中，unit 表示为 uint256 的别名。

```
uint256 x;  
uint y;  
int256 z;
```

这一类型的数据也可以用 constant 关键字来声明，这意味着编译器不会为这些变量保留任何存储。期间，变量每次出现都将被替换为实际值。例如：

```
uint constant z=10+10;
```

状态变量在函数的主体之外声明，只要合约一直存在，状态变量在整个合约中都是

可用的。当然，这也取决于所赋予的可访问性权限。

3. address

该数据类型定义为 160 位长（20 字节）的数据值，同时包含多个成员可与合约进行交互和查询，如下所示：

- ❑ **balance**。该成员返回 Wei 中地址的余额。
- ❑ **send**。该成员用于向某一地址（以太坊中的 160 位地址）发送一定量的 Ether，并根据交易结果返回 true 或 false。例如：

```
address to = 0x6414cc08d148dce9ebf5a2d0b7c220ed2d3203da;  
address from = this;  
if (to.balance < 10 && from.balance > 50) to.send(20);
```

- ❑ **调用函数**。call、callcode 和 delegate atecall 旨在与缺少 ABI（应用程序二进制接口）的函数进行交互。这些功能应该谨慎使用，并会对类型安全和合约安全产生负面影响。
- ❑ **数组值类型**（固定尺寸或动态尺寸的字节数组）。Solidity 中定义了固定尺寸和动态尺寸的字节数组。其中，固定尺寸关键字其范围从 bytes1 到 bytes32；而动态尺寸关键字包括字节和字符串。byte 用于原始字节数据，字符串则用于 UTF-8 编码的字符串。当此类数组通过值返回时，调用时将产生 gas 消费。另外，length 是数组值类型的成员，并返回字节数组的长度。

静态数组（固定尺寸）示例如下所示：

```
bytes32[10] bankAccounts;
```

动态尺寸数组如下所示：

```
bytes32[] trades;
```

length 的获取方式如下所示：

```
trades.length;
```

8.3.2 字面值

字面值常用于表示固定值。

1. 整数字面值

整数字面值定义为 0~9 范围内的十进制数序列，对应示例如下所示：

```
uint8 x = 2;
```

2. 字符串字面值

字符串字面值指定一组用双引号或单引号定义的字符，如下所示：

```
'packt'
"packt"
```

3. 十六进制字面值

十六进制字面值用关键字 `hex` 作为前缀，并在双引号或单引号内指定，如下所示：

```
(hex'AABBCC');
```

8.3.3 枚举值

枚举值支持用户自定义的类型，如下所示：

```
enum Order{Filled, Placed, Expired };
Order private ord;
ord=Order.Filled;
```

所有整数类型的显式转换都可以使用 `enum`。

8.3.4 函数类型

函数包括两种类型，即内部函数和外部函数。

❑ 内部函数。此类函数仅可用于当前合约环境下。

❑ 外部函数。此类函数可通过外部函数被调用。

Solidity 中的函数可以被标记为常数。常数函数不能改变合约中的任何内容，仅在被调用时返回值，并且不消费任何 `gas`。这也是在第 7 章讨论的调用概念的实际实现。

函数声明语法如下所示：

```
function <nameofthefunction> (<parameter types> <name of the variable>)
{internal|external} [constant] [payable] [returns (<return types>
<name of the variable>)]
```

8.3.5 引用类型

顾名思义，此类型通过引用进行传递，下面将对此逐一讨论。

1. 数组

数组表示一组彼此相邻的、相同大小和类型的元素，且位于内存中的某个位置处。

这一概念与其他编程语言并无太多差异。除此之外，数组中包含两个名为 `length` 和 `push` 的成员。数组的对应示例如下所示：

```
uint[] OrderIds;
```

2. 结构

结构可在某一逻辑分组中定义一组不同的数据类型，进而可以用来定义新的数据类型，如下所示：

```
Struct Trade
{
    uint tradeid;
    uint quantity;
    uint price;
    string trader;
}
```

3. 数据位置

数据位置指定了特定复杂数据类型的存储位置。根据默认行为或注释内容，对应位置可以是存储库或内存。该机制适用于数组和结构，对此，可以使用 `storage` 或 `memory` 关键字定义。由于内存和存储之间的复制行为常会占用较大的开销，所以指定一个位置可能有助于控制 `gas` 的消费。`calldata` 是用于存储函数参数的另一个内存位置。相应地，外部函数的参数将使用 `calldata` 内存。默认情况下，函数的参数存储在 `memory` 中，而所有其他本地变量则使用 `storage`。另一方面，状态变量则需要使用 `storage`。

8.3.6 映射

映射一般指键-值映射，并可视作一种将值与键关联的方法。映射的全部值均已初始化为 0，例如：

```
mapping (address => uint) offers;
```

其中，`offers` 声明为一个映射。另一个相对清楚的例子如下所示：

```
mapping (string => uint) bids;
bids["packt"] = 10;
```

这基本上可视为一个字典或哈希表，其中字符串值被映射为整数值。此处，名为 `bid` 的映射包含了 `packt` 字符串值，并映射为数值 10。

8.3.7 全局变量

Solidity 提供了一些全局变量，并在全局名称空间中可用。这一类变量提供了与区块和交易相关的信息。此外，Solidity 还提供了加密函数和与地址相关的变量。

可用函数和变量的子集如下所示：

```
keccak256(...) returns (bytes32)
```

该函数用于计算提供给函数参数的 keccak256 哈希值。

```
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)
```

该函数返回源自椭圆曲线公钥的关联地址。

```
block.number
```

这将返回当前区块号。

8.3.8 控制结构

Solidity 中定义了 if-else、do、while、for、break、continue、return 等控制结构，其工作方式与 C 语言或 JavaScript 类似。

1. 事件

Solidity 中的事件可以用来记录 EVM 日志中的某些事件。当需要将合约中的任何变化或事件通知与外部接口时，事件机制非常有用。日志存储在交易日志的区块链中，且无法从合约中访问日志，而是作为一种机制通知合约中状态的更改或事件的发生（满足某项条件）。

在下列简单示例中，如果传递给 function Matcher 的 x 参数等于或大于 10，那么 valueEvent 事件将返回 true。

```
contract valueChecker {
    uint8 price=10;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) returns (bool)
    {
        if (x>=price)
        {
            valueEvent(true);
        }
    }
}
```



```
        return true;
    }
}
}
```

2. 继承

Solidity 支持继承机制。关键字 `is` 用于派生合约。在下面的例子中，`valueChecker2` 继承自 `valueChecker` 合约。另外，派生的合约可以访问父合约中的所有非私有成员。

```
contract valueChecker
{
    uint8 price=10;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) returns (bool)
    {
        if (x>=price)
        {
            valueEvent(true);
            return true;
        }
    }
}

contract valueChecker2 is valueChecker
{
    function Matcher2() returns (uint)
    {
        return price + 10;
    }
}
```

在上述示例中，如果将“`uint8 price = 10`”修改为“`uint8 private price = 10`”，那么 `valueChecker2` 合约将无法访问该变量，其原因在于，该成员声明为私有，其他合约无法对其予以访问。

3. 库

库只在特定的地址部署一次，其代码通过 EVM 的 `CALLCODE` / `DELEGATECALL` 操作码调用。库的关键思想是代码可重用性。库合约十分类似，并可作为基础型合约。库的声明示例如下所示：

```
library Addition
{
    function Add(uint x,uint y) returns (uint z)
```

```
{
    return x + y;
}
```

该库可以在合约中调用。首先，需要导入该库，随后即可在代码中的任何地方使用。对应示例如下所示：

```
Import "Addition.sol"
function Addtwovalues() returns(uint)
{
    return Addition.Add(100,100);
}
```

库也存在一些限制因素。例如，库无法定义状态变量，且不支持继承机制。此外，库也无法接收 Ether，这与可接收 Ether 的合约形成了鲜明的对比。

4. 函数

Solidity 中的函数定义为与合约相关联的代码模块。函数声明包括名称、可选参数、访问修饰符、可选常量关键字和可选返回类型，如下所示：

```
function orderMatcher(uint x) private constant returns(bool returnvalue)
```

其中，**function** 是用来声明函数的关键字；**orderMatcher** 表示为函数名；**uint x** 定义为一个可选参数；**private** 表示访问修饰符，并以此控制源自外部合约的函数访问；**constant** 是一个可选关键字，表明该函数不会修改合约中的任何内容，且只用于从合约中获取数据值；**return(bool returnvalue)** 则是可选的函数返回类型。

□ 函数的定义方式。函数的定义语法如下所示：

```
function <name of the function>(<parameters>) <visibility
specifier> returns (<return data type> <name of the variable>)
{
    <function body>
}
```

□ 函数签名。Solidity 中的函数通过签名标识，即完整签名字符串的 keccak-256 哈希值的前 4 个字节，且在浏览器的 Solidity 中可见，如图 8.27 所示。其中，D99c89cb 表示为函数名 **Matcher** 的 32 字节 keccak - 256 哈希值的前 4 个字节。

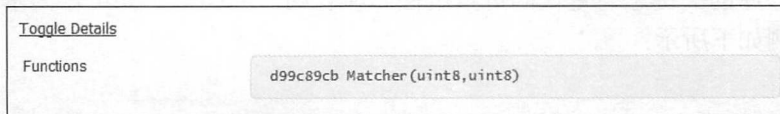


图 8.27 浏览器 Solidity 中显示的函数哈希值

在该示例函数中，Matcher 的签名哈希值为 d99c89cb，此类信息对于构建接口非常有用。

- ❑ 函数的输入参数。函数的输入参数以<数据类型><参数名>这一形式声明。下列示例表明，uint x 和 uint y 是 checkValues 函数的输入参数。

```
contract myContract
{
  function checkValues(uint x, uint y)
  {
  }
}
```

- ❑ 函数的输出参数。函数的输出参数以< data type> <参数名>这一形式声明。下列例子显示了一个返回 uint 值的简单函数。

```
contract myContract
{
  Function getValue() returns (uint z)
  {
    z=x+y;
  }
}
```

函数可以返回多个值。在上述函数中，getValue 只返回一个值。但是，一个函数可以返回多达 14 个不同数据类型的值。可以视具体情况省略未使用的返回参数的名称。

- ❑ 内部函数调用。在当前合约的上下文中，函数可于内部直接被调用，进而调用同一合约中现有的函数。此类调用将在 EVM 字节码级别上进行简单的跳转操作。
- ❑ 外部函数调用。外部函数调用是通过消息机制并在合约间进行的。在这种情况下，所有的函数参数都被复制到内存中。如果内部函数调用使用了 this 关键字，那么该函数也被认为是一个外部调用。this 变量表示指向当前合约的指针，并可以显式转换为地址，合约的所有成员均源自该地址。
- ❑ 回调函数。在合约中，回调函数定义为一个未命名的函数，且不包含参数和返回数据。该函数在每次接收 Ether 时执行。如果合约须获取 Ether，则需要在合约中实现该函数；否则将抛出异常并返回 Ether。如果合约中不存在可匹配的其他函数签名，该函数也将执行。另外，如果可获取 Ether，则应该用 payable 修饰符声明该函数。此处，payable 不可缺少；否则该函数将无法接收 Ether。该函数可使用 address.call() 方法予以调用，例如：

```
function ()
{
```

```
throw;  
}
```

若 `fallback` 函数根据前述条件被调用，则会调用 `throw`，并回滚至调用前的状态。除 `throw` 之外，也存在一些其他结构。例如，可记录某项事件，并用作警告消息，以反馈至应用程序的调用结果。

- ❑ 修饰符函数。此类函数用于调整函数的行为，并可在其他函数之前调用。通常情况下，在执行函数之前，修饰符函数用于检查某些条件或验证结果。另外，函数中还使用了 `_`（下划线），当修饰符被调用时，将替换为该函数的实际主体。基本上，该机制定义了需要保护的函数，这一概念类似于其他语言中的 `guard` 函数。
- ❑ 构造函数。作为可选函数，构造函数具有与合约相同的名称，并在创建合约时执行一次。构造函数不可被用户调用，而且合约中仅允许存在一个构造函数。这也意味着无法提供重载功能。
- ❑ 函数可见性说明符（访问修饰符）。函数可通过以下 4 种访问修饰符加以定义。
 - **External**：此类函数可从其他合约或交易中被访问，且无法在内部进行调用，除非使用 `this` 关键字。
 - **Public**：默认状态下，函数均处于公有状态，并可在内部或利用消息机制予以调用。
 - **Internal**：内部函数仅相对于父合约派生的合约可见。
 - **Private**：私有函数仅相对于声明内的同一合约可见。
- ❑ 其他重要的关键字/函数 `throw`。`throw` 用于终止当前执行。最终，所有状态变化均被恢复。在这种情况下，没有 `gas` 返回至交易发起者，所有剩余的 `gas` 均已被消耗。

5. Solidity 源代码文件的设置

- ❑ 版本和 `pragma`。为了解决 Solidity 编译器版本可能出现的兼容性问题，`pragma` 可以用来指定兼容编译器的版本，例如：

```
pragma solidity ^0.5.0
```

这将确保源文件不会被小于 0.5.0 的版本所编译，且版本应从 0.6.0 开始。

- ❑ **Import**：在 Solidity 中，**Import** 允许将现有的 Solidity 文件符号导入至当前全局范围，类似于 JavaScript 中的导入语句，例如：

```
Import "module-name";
```

- ❑ **注释**。Solidity 源代码文件中的注释方式类似于 C 语言：多行注释包含在 `/*`

和 “*/” 中，而单行注释则从 “//” 开始。

图 8.28 显示了包含 pragma、import 以及注释的 Solidity 程序示例。

```
1 pragma solidity ^0.4.0; //specify compiler version
2- /*
3 This is a simple value checker contract
4 that checks the value provided and returns boolean value
5 based on the condition expression evaluation
6 */
7 import "dev.oraclize.it/api.sol";
8- contract valueChecker {
9     uint price=10;
10    // This is price variable decared and initialized with value 10.
11    event valueEvent(bool returnValue);
12    function Matcher (uint8 x) returns (bool)
13-    {
14        if (x>=price)
15-        {
16            valueEvent(true);
17            return true;|
18        }
19    }
20 }
```

图 8.28 Solidity 示例程序

Solidity 语言具有丰富的功能，且仍处于不断完善的过程中；同时还包含了详细的在线文档和编码指南。

8.4 引入 Web3

Web3 是一个 JavaScript 库，可以通过 RPC 通信进而与以太坊节点通信。Web3 通过显示于 RPC 上的方法来工作，因而支持基于 Web3 库的用户界面开发，以便与部署在区块链上的合约进行交互。

下列命令可通过 geth 显示相关方法：

```
$ geth --datadir .ethereum/privatenet/ --networkid 786 --rpc --rpcapi 'web3,eth,debug' --rpcport 8001 --rpccorsdomain 'http://localhost:7777'
```

需要注意的是，--rpcapi 标记支持 Web3、eth 和 debug 方法。

Web3 是一个功能强大的库，可以通过绑定一个 geth 实例予以深入考察。后续章节将介绍基于 JavaScript /HTML 前端的 Web3 技术。

geth 可通过下列命令绑定：

```
$ geth attach ipc:.ethereum/privatenet/geth.ipc
```

当 geth JavaScript 处于运行状态时，即可对 Web3 进行查询，如图 8.29 所示。

```

> web3.version
{
  api: "0.19.3",
  ethereum: "0x3f",
  network: "786",
  node: "Geth/v1.9.2-stable-c8695209/linux/go1.7.3",
  whisper: undefined,
  getEthereum: function(callback),
  getNetwork: function(callback),
  getNode: function(callback),
  getWhisper: function(callback)
}
>

```

图 8.29 基于 geth 的 Web3

简单的合约可以使用 `geth` 进行部署，并通过 `geth` 提供的命令行接口（控制台或绑定）使用 `Web3` 进行交互。例如，可使用下列源代码：

```

pragma solidity ^0.4.0;
contract valueChecker {
  uint price=10;
  event valueEvent(bool returnValue);
  function Matcher (uint8 x) returns (bool)
  {
    if (x>=price)
    {
      valueEvent(true);
      return true;
    }
  }
}

```

开启 `geth` 控制台并执行下列操作步骤：

(1) 声明名为 `simplecontractsource` 的变量，并将程序代码附于其中：

```

> var simplecontractsource = "pragma solidity ^0.4.0; contract
valueChecker { uint price=10;event valueEvent(bool returnValue);
function Matcher (uint8 x) returns (bool) { if (x>=price)
{valueEvent(true); return true; } } }"

```

这将显示如下输出结果：

```
undefined
```

注意，源代码需要位于一行中，也就是说不包含换行符。可以使用以下命令在 Linux 中实现这一点：

```
$ tr --delete '\n' < valuechecker.sol > valuecheckersingleline.sol
```


在上述示例中，文件 `valuechecker.sol` 中包含了换行符 `\n`，而 `valuecheckersingleline.sol` 则是删除文件中换行符后产生的输出文件。随后，可以将代码从文件复制粘贴到 `geth JavaScript` 控制台。

(2) 验证 Solidity 是否有效，如下所示：

```
> eth.getCompilers()
["Solidity"]
```

(3) 定义一个变量，并通过 Solidity 分配和编译代码，如下所示：

```
> var
simplecontractcompiled=eth.compile.solidity(simplecontractsource)
undefined
```

(4) 输入 `simplecontractcompiled`，这将显示如图 8.30 所示的结果。此处，`simplecontractcompiled` 已在步骤 (3) 中赋予数据。

```
> simplecontractcompiled
{
  valueChecker: {
    code: "0x6060604052600a60005534610000575b60378061001c6000396000f3606060405260e060020a60003
50463f9d55e218114601c575b6000565b346000576029600435603d565b60408051911515825251908190036020019
0f35b6000805460ff831610608157604080516001815290517f3eb1a229ff7995457774a4bd31ef7b13b6f4491ad1e
bb8961af120b8b4b6239c9181900360200190a15060015b5b91905056",
    info: {
      abiDefinition: [{...}, {...}],
      compilerOptions: "--combined-json bin,abi,userdoc,devdoc --add-std --optimize",
      compilerVersion: "0.4.6",
      developerDoc: {
        methods: {}
      },
      language: "Solidity",
      languageVersion: "0.4.6",
      source: "pragma solidity ^0.4.0; contract valueChecker { uint price=10; event valueEvent
(bool returnValue); function Matcher (uint8 x) returns (bool) { if (x>=price) { valueEvent(tru
e); return true; } } }",
      userDoc: {
        methods: {}
      }
    }
  }
}
```

图 8.30 `simplecontractcompiled` 输出

(5) 定义一个变量，并与下列内容交互：

```
>var simplecontractinteractor=eth.contract
(simplecontractcompiled.valueChecker.info.abiDefinition);
undefined
```

(6) 检测 ABI（应用程序二进制接口），如下所示：

```
> simplecontractinteractor.abi
[
  {
    constant: false,
    inputs: [
      {
        name: "x",
        type: "uint8"
      }
    ],
    name: "Matcher",
    outputs: [
      {
        name: "",
        type: "bool"
      }
    ],
    payable: false,
    type: "function"
  },
  {
    anonymous: false,
    inputs: [
      {
        indexed: false,
        name: "returnValue",
        type: "bool"
      }
    ],
    name: "valueEvent",
    type: "event"
  }
]
```

(7) 以十六进制格式检查 valueChecker 代码，如下所示：

```
> simplecontractcompiled.valueChecker.code
```

这将生成下列结果（实际内容可能稍有不同）：

```
"0x6060604052600a60005534610000575b60878061001c6000396000f360606040526
0e060020a6000350463f9d55e218114601c575b6000565b346000576029600435603d5
65b604080519115158252519081900360200190f35b6000805460ff831610608157604
080516001815290517f3eb1a229ff7995457774a4bd31ef7b13b6f4491ad1ebb8961af
120b8b4b6239c9181900360200190a15060015b5b91905056"
```

(8) 输入下列代码段。注意，数据字段包含了针对 simplecontractcompiled 的代码。

```
>var simplecontractTransaction = simplecontractinteractor.new({
  from: eth.coinbase,
  data: simplecontractcompiled.valueChecker.code,
  gas: 2000000
},
function(err, contract) {
```



```
    if (err) {  
      console.error(err);  
    } else {  
      console.log(contract);  
      console.log(contract.address);  
    }  
  });
```

若返回系列错误消息:

```
Error: account is locked  
Undefined
```

则需解锁当前账户。对此, 首先可通过下列命令获得账户 ID:

```
> personal.listAccounts  
["0x76f11b383dbc3becf8c5d9309219878caae265c3",  
"0xcce6450413ac80f9ee8bd97ca02b92c065d77abc"]
```

针对解锁账户输入下列命令:

```
> personal.unlockAccount  
("0x76f11b383dbc3becf8c5d9309219878caae265c3")  
Unlock account 0x76f11b383dbc3becf8c5d9309219878caae265c3
```

输入当前账户的密码:

```
Passphrase:  
true
```

待账户解锁后再次输入上述代码。此时, 错误消息将显示为:

```
> Error: The contract code couldn't be stored, please check your  
gas amount.
```

对此, 可尝试增加 gas 量。如果输入的 gas 量值过大, 将显示如下所示的错误消息:

```
Error: Exceeds block gas limit  
undefined
```

(9) 一旦账户成功解锁, 即可启动矿工并挖掘合约 (无须对当前账户解锁进而启动挖掘过程。账户解锁的必要性体现在, 挖掘合约并在区块链上生成该合约), 如下所示:

```
> miner.start()  
true
```

当合约正确生成后，将显示下列输出结果：

```
[object Object]
undefined
undefined
> [object Object]
0x94a1107f2585f0ab931c71f2f8f02e9f5ab888c0
```

在合约被挖掘后，这将显示新合约的地址。

(10) 为了方便地与合约交互，合约地址可赋予至某个变量中，如下所示：

```
> var simplecontractaddress=
"0x94a1107f2585f0ab931c71f2f8f02e9f5ab888c0"
Undefined
```

(11) 其中涵盖了大量的方法，进而可对合约实施进一步的查询，例如：

```
> var deployedaddress=eth.getCode(simplecontractaddress);
undefined
> deployedaddress
"0x606060405260e060020a6000350463f9d55e218114601c575b6000565b346000
576029600435603d565b604080519115158252519081900360200190f35b6000805
460ff831610608157604080516001815290517f3eb1a229ff7995457774a4bd31ef
7b13b6f4491ad1ebb8961af120b8b4b6239c9181900360200190a15060015b5b919
05056"
> eth.getBalance(simplecontractaddress)
0
```

(12) 此后，可创建名为 simplecontractinstance 的对象，并用于调用此类方法，如下所示：

```
simplecontractinstance = web3.eth.contract(simplecontractcompiled
.valueChecker.info.abiDefinition).at(simplecontractaddress);
```

(13) 相关方法列表如下所示：

```
> simplecontractinstance.Matcher.
simplecontractinstance.Matcher.apply
simplecontractinstance.Matcher.constructor
simplecontractinstance.Matcher.request

simplecontractinstance.Matcher.arguments
simplecontractinstance.Matcher.estimateGas
simplecontractinstance.Matcher.sendTransaction
```



```

simplecontractinstance.Matcher.bind
simplecontractinstance.Matcher.getData
simplecontractinstance.Matcher.toString

simplecontractinstance.Matcher.call
simplecontractinstance.Matcher.length
simplecontractinstance.Matcher.uint8

simplecontractinstance.Matcher.caller
simplecontractinstance.Matcher.prototype

```

(14) 相应地, 合约也可执行进一步查询。在下列示例中, Matcher 函数通过参数被调用。回忆一下, 在代码中, 可设置某项条件并进行检测: 如果某个值大于或等于 10, 则返回 true; 否则返回 false。如下所示:

```

> simplecontractinstance.Matcher.call(12)
true
> simplecontractinstance.Matcher.call(9)
false
> simplecontractinstance.Matcher.call(0)
false
> simplecontractinstance.Matcher.call(12)
true

```

8.4.1 POST 请求

HTTP 上的 jsonrpc 还可与 geth 进行交互。对此, 可采用 curl。下面将通过一些示例了解 POST 请求, 以及如何使用 curl 生成 POST 请求。另外, 读者可访问 <https://curl.haxx.se/> 下载 curl。

在具体使用 HTTP 上的 JsonRPC 接口之前, 应按照下列方式启动 geth:

```
--rpcapi web3
```

这将在 HTTP 上开启 Web3。

Linux 命令 curl 可用于 HTTP 上的通信机制, 稍后将对此予以展示。例如, 当采用 personal_listAccounts 方法获取账户列表时, 可使用下列命令:

```

$ curl --request POST --data
'{"jsonrpc": "2.0", "method": "personal_listAccounts", "params":
[[], "id": 4}]' localhost:8001

```

这将返回包含账户列表的 JSON 对象，如下所示：

```
{"jsonrpc": "2.0", "id": 4, "result":  
[  
  "0x76f11b383dbc3becf8c5d9309219878caae265c3",  
  "0xcce6450413ac80f9ee8bd97ca02b92c065d77abc"]  
]}
```

在上述 curl 命令中，--request 用于指定请求命令，POST 即为当前请求；--data 用于指定参数和数值；最后，localhost:8001 表示为 geth 中开启 HTTP 端点之位置。

8.4.2 HTML 和 JavaScript 前端

合约应通过 Web 页面并采用友好方式进行交互，对此，可采用包含 HTML/JS/CSS 页面的 web3.js 库。目前 HTTP Web 服务器均支持 HTML 内容，而 web3.js 可通过本地 RPC 连接至处于原先状态的以太坊客户端（geth），并向区块链上的合约提供一个接口，对应结构如图 8.31 所示。

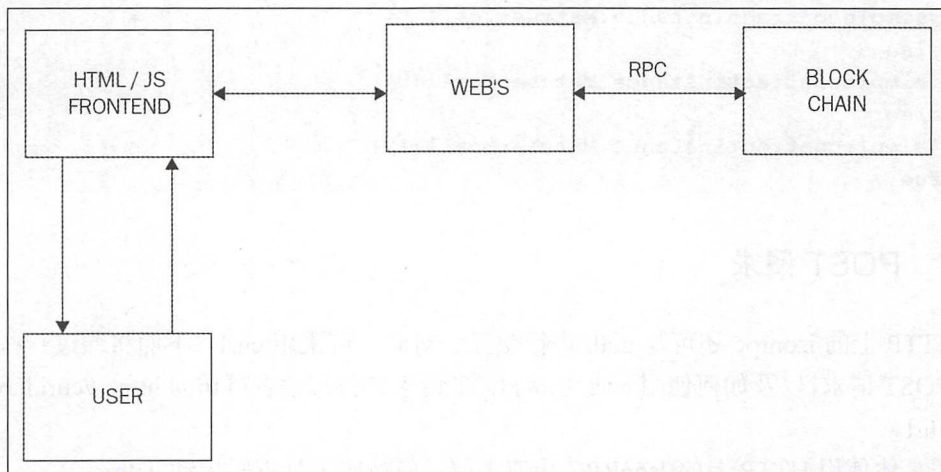


图 8.31 web3.js、前端和区块链交互式体系结构

1. 安装 web3.js

Web3 可通过 npm 以及下列命令进行安装。

```
$ npm install web3
```

或者访问 <https://github.com/ethereum/web3.js> 直接下载。

通过 npm 下载的 web3.min.js 可在 HTML 文件中被引用，且位于 node_modules 中。例如，根据具体情况，该文件可复制至主应用程序所处的目录中，并以此为起点加以使

用。当该文件在 HTML 或 JS 被成功引用后, Web3 须提供 HTTP 运营商进而被初始化。通常,这可视为 geth 客户端显示的本地主机 HTTP 端点链接,并可通过下列命令实现:

```
web3.setProvider(new web3.providers.HttpProvider('http://localhost:8001'));
```

待供应商设置完毕后,即可通过 Web3 对象及其相关方法实现合约和区块链的交互操作。

相应地, Web3 对象可采用下列命令创建:

```
if (typeof web3 !== 'undefined')
{
    web3 = new Web3(web3.currentProvider);
}
else
{
    web3 = new Web3(new
    Web3.providers.HttpProvider("http://localhost:8001"));
}
```

下面通过一个示例展示 web3.js 应用,并通过 HTTP Web 服务器上的网页实现合约间的交互。相关步骤如下:

- (1) 在 home 目录中创建名为/simplecontract/app 的目录。
- (2) 创建名为 simplecontractcompiled.js 的文件,如下所示:

```
simplecontractcompiled={
valueChecker: {
code:
"0x6060604052600a60005534610000575b60878061001c6000396000f360606040
5260e060020a6000350463f9d55e218114601c575b6000565b34600057602960043
5603d565b604080519115158252519081900360200190f35b6000805460ff83161
0608157604080516001815290517f3eb1a229ff7995457774a4bd31ef7b13b6f449
1ad1ebb8961af120b8b4b6239c9181900360200190a15060015b5b91905056",
info:
{
abiDefinition:
[{}
constant: false,
inputs:
[{}
name: "x",
type: "uint8"
```

```
    }],  
    name: "Matcher",  
    outputs:  
    [{  
      name: "",  
      type: "bool"  
    }],  
    payable: false,  
    type: "function"  
  },  
  {  
    anonymous: false,  
    inputs:  
    [{  
      indexed: false,  
      name: "returnValue",  
      type: "bool"  
    }],  
    name: "valueEvent",  
    type: "event"  
  }],  
  compilerOptions: "--combined-json bin,abi,userdoc,devdoc --addstd  
--optimize", compilerVersion: "0.4.6",  
  developerDoc:  
  {  
    methods: {}  
  },  
  language: "Solidity",  
  languageVersion: "0.4.6",  
  source: "pragma solidity ^0.4.0; contract valueChecker { uint  
price=10; event valueEvent(bool returnValue);  
function Matcher (uint8 x) returns (bool) { if (x>=price) {  
valueEvent(true); return true; } } }",  
  userDoc: {  
    methods: {}  
  }  
}
```

该文件包含了各种元素，其中较为重要的是 ABI（应用程序二进制接口），并可通过 `geth` 进行查询。

(3) 创建名为 `simplecontract.js` 的文件，如下所示：

```
if (typeof web3 !== 'undefined')
{
  web3 = new Web3(web3.currentProvider);
}
else
{
  web3 = new Web3(new
    Web3.providers.HttpProvider("http://localhost:8001"));
}
console.log("Coinbase: " + web3.eth.coinbase);
var simplecontractaddress = "0x94a1107f2585f0ab931c71f2f8f02e9
  f5ab888c0";

simplecontractinstance =
  web3.eth.contract(simplecontractcompiled.valueChecker
    .info.abiDefinition).at(simplecontractaddress);
var code = web3.eth.getCode(simplecontractaddress);
console.log("Contract balance: " +
  web3.eth.getBalance(simplecontractaddress));
console.log("simple contract code" + code);
function callMatchertrue()
{
  var txn = simplecontractinstance.Matcher.call(12);{
};
console.log("return value: " + txn);
}
function callMatcherfalse()
{
  var txn = simplecontractinstance.Matcher.call(1);{
};
console.log("return value: " + txn);
}
```

此文件是包含创建 Web3 对象代码的主要 JavaScript 文件，还提供了用于与区块链合约交互的方法。

2. 创建 Web3 对象

Web3 对象的创建过程如下列代码所示：

```
if (typeof web3 !== 'undefined')
{
  web3 = new Web3(web3.currentProvider);
```

```
}  
else  
{  
    web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:  
        8001"));  
}
```

其中，代码首先检测是否存在运行商。若是，则将其设置为当前运营商；否则，则将 Web3 供应商设置为 localhost: 8001，这也是本地 geth 实例所运行之处。

另外，还可通过调用 Web3 方法检测有效性，如下所示：

```
console.log("Coinbase: " + web3.eth.coinbase);
```

代码简单地使用 console.log 输出 coinbase，即调用 web3.eth.coinbase 方法。调用成功后，Web 对象即被正确地创建，并可得到 HttpProvider。除此之外，其他调用也可用于验证有效性，但出于简单性考虑，此处使用了 web3.eth.coinbase。

当向某个变量赋予地址时，可采用下列代码：

```
var simplecontractaddress = "0x94a1107f2585f0ab931c71f2f8f02e9f5ab888c0";
```

上述语句赋予了部署于区块链上的合约地址值，操作成功后，simplecontractaddress 将包含当前合约的地址。这也是前述示例中步骤（9）中生成的地址值，此时合约已发布。当前代码只是简单地使用了该地址值。

下列代码负责生成主合约对象。

```
simplecontractinstance = web3.eth.contract(simplecontractcompiled  
    .valueChecker.info.abiDefinition)  
    .at(simplecontractaddress);
```

上述代码片段将生成一个对象，以供后续代码使用，进而与区块链上的合约进行交互。simplecontractinstance 将显示合约函数；web3.eth.contract 接收 ABI 数组作为参数，并通过 simplecontractcompiled.valueChecker.info.abiDefinition 予以传递；最后，.at 接收合约地址作为参数。

下列示例显示了如何获取合约地址代码（可选）。

```
var code = web3.eth.getCode(simplecontractaddress);  
console.log("simple contract code" + code);
```

上述语句用于查询合约代码，仅简单地调用了 web3.eth.getCode，并接收区块链上的合约地址作为参数。最后，console.log 用于输出合约代码（输出代码变量）。

下列代码显示了合约余额。


```
console.log("Contract balance:"  
+web3.eth.getBalance(simplecontractaddress));
```

上述代码将调用 `web3.eth.getBalance`，接收合约地址作为参数，并输出合约的余额。当前余额值为 0。

一旦 Web3 对象被正确创建，将生成 `simplecontractinstance`，即可方便地执行合约函数调用，如下所示：

```
function callMatchertrue()  
{  
  var txn = simplecontractinstance.Matcher.call(12);{  
};  
  console.log("return value: " + txn);  
}  
  
function callMatcherfalse()  
{  
  var txn = simplecontractinstance.Matcher.call(1);{  
};  
  console.log("return value: " + txn);  
}
```

上述代码通过 `simplecontractinstance.Matcher.call` 实现了调用，随后传递参数值。回忆一下。Solidity 代码的 `Matcher` 函数如下所示：

```
function Matcher (uint8 x) returns (bool)
```

该函数接收 `uint8` 类型的参数 `x`，并返回一个布尔值（`true` 或 `false`）。相应地，针对当前合约执行调用，如下所示：

```
var txn = simplecontractinstance.Matcher.call(12);
```

其中，`console.log` 用于输出函数调用的返回值。一旦调用结果出现于 `txn` 变量中，即可用于程序中的各处，例如，可作为另一个 JavaScript 函数的参数。

最后名为 `index.html` 的 HTML 文件可通过下列代码生成：

```
<html>  
<head>  
  <title>SimpleContract Interactor</title>  
  <script src="./web3.min.js"></script>  
  <script src="./simplecontractcompiled.js"></script>  
  <script src="./simplecontract.js"></script>  
</head>
```

```
<body>
  <button onclick="callMatchertrue()">callTrue</button>
  <button onclick="callMatcherfalse()">callFalse</button>
</body>
</html>
```

这里，建议运行适当的 Web 服务器以处理 HTML 内容（例如 index.html）。或者，文件也可以通过文件系统浏览，但对于一些较大项目，这可能会导致某些问题。作为一种良好的实践体验，应始终使用 Web 服务器。可以使用如图 8.32 所示的命令启动 Python 中的一个快速 Web 服务器。该服务器将从已运行的相同目录中处理 HTML 内容。实际上，Python 并非必要，甚至可以采用 Apache 服务器或任何其他 Web 容器。

```
imran@drequinox-0P7010:~/simplecontract/app$ python -m SimpleHTTPServer 7777
Serving HTTP on 0.0.0.0 port 7777 ...
```

图 8.32 Python 中简单的服务器

当前，任何浏览器都可查看服务于 TCP 端口 7777 的 Web 页面，如图 8.33 所示。需要注意的是，输出内容位于浏览器的控制台窗口中，因而须启用浏览器的控制台以查看输出。

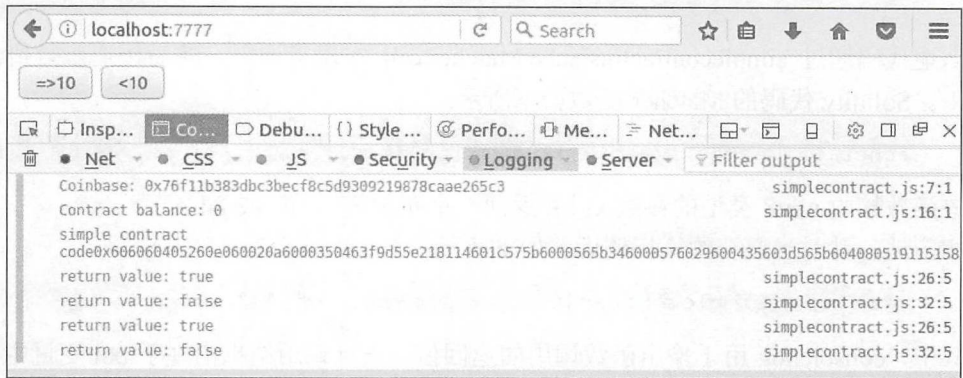


图 8.33 与合约进行交互

出于简单考虑，代码中的数值采用了硬编码，并在 index.html 文件中生成了两个按钮，调用包含应编码值的函数。该过程参数可通过 Web3 传递至合约中，并返回相应值。

在上述按钮背后，存在两个被调函数。其中，callMatchertrue 方法包含一个硬编码值 12，并通过下列代码传递至合约。

```
simplecontractinstance.Matcher.call(12)
```


返回值通过下列代码在控制台中输出，代码首先调用 `Matcher` 函数，随后将数值赋予 `txn` 变量，稍后在控制台上予以输出。

```
simplecontractinstance.Matcher.call(1)
function callMatchertrue()
{
  var txn = simplecontractinstance.Matcher.call(12);{
};
console.log("return value: " + txn);
}
```

类似地，`callMatcherfalse()` 函数将硬编码值 1 传递至合约中，如下所示：

```
simplecontractinstance.Matcher.call(1)
```

随后输出返回值，如下所示：

```
console.log("return value: " + txn);
function callMatcherfalse()
{
  var txn = simplecontractinstance.Matcher.call(1);{
};
console.log("return value: " + txn);
}
```

当前示例展示了 Web3 库与区块链中合约之间的交互方式。

8.4.3 开发框架

当前存在多种用于以太坊的开发框架。如前所述，常规的手动方式部署契约通常较为耗时；而某些框架可简化这一操作过程，例如 `Truffle` 和 `embark`。其中，`Truffle` 则是应用最为广泛的框架。

`Truffle` 作为一类开发环境，可简化以太坊合约的部署。当与基于 `Mocha` 和 `Chai` 的自动化测试框架结合使用时，`Truffle` 提供了合约的编译和链接功能，另外还可方便地将合约部署至 `PrivateNet`、公有或 `TestNet` 以太坊区块链中。除此之外，`Truffle` 还提供了资源管线（`asset pipeline`），进而简化所处理的 `JavaScript` 文件，以供浏览器使用。

1. 安装 Truffle

在安装之前，假定节点可用且可被查询。如果节点不可用，则首先需要安装节点，进而安装 `Truffle`，如下所示：

```
drequinox@drequinox-OP7010:~/testdapp$ nodejs --version
v7.2.1
drequinox@drequinox-OP7010:~/testdapp$ node --version
v7.2.1
```

Truffle 的安装过程十分简单，可在 npm 的基础上使用下列命令：

```
$ sudo npm install -g truffle
```

该过程将占用少量时间，待安装完毕后，Truffle 可用于显示帮助信息，以确保已被正确安装，如图 8.34 所示。

```
drequinox@drequinox-OP7010:~/testdapp$ truffle
Truffle v2.1.1 - a development framework for Ethereum

Usage: truffle [command] [options]

Commands:
  build          => Build development version of app
  compile        => Compile contracts
  console        => Run a console with deployed contracts instantiated and available (REPL)
  create:contract => Create a basic contract
  create:migration => Create a new migration marked with the current timestamp
  create:test     => Create a basic test
  exec           => Execute a JS file within truffle environment
  init           => Initialize new Ethereum project, including example contracts and tests
  list           => List all available tasks
  migrate        => Run migrations
  networks       => Show addresses for deployed contracts on each network
  serve          => Serve app on localhost and rebuild changes as needed
  test           => Run tests
  version        => Show version number and exit
  watch          => Watch filesystem for changes and rebuild the project automatically
```

图 8.34 Truffle 帮助信息

另外，还可访问 <https://github.com/ConsenSys/truffle> 获取存储库，在本地复制后即可安装 Truffle。通过下列命令，可利用 Git 复制存储库。

```
https://github.com/ConsenSys/truffle.git
```

当初初始化 Truffle 时，首先须对当前项目创建一个目录，如下所示：

```
mkdir testdapp
```

随后，可修改为 testdapp 并运行下列命令：

```
~/testdapp$ truffle init
```

当成功执行命令后，将创建如图 8.35 所示的目录结果，并可通过 Linux 中的 tree 命令进行查看。


```
drequinox@drequinox-OP7010:~/testdapp$ tree
├── app
│   ├── images
│   ├── index.html
│   ├── javascripts
│   │   └── app.js
│   └── stylesheets
│       └── app.css
├── contracts
│   ├── ConvertLib.sol
│   ├── MetaCoin.sol
│   └── Migrations.sol
├── migrations
│   ├── 1_initial_migration.js
│   └── 2_deploy_contracts.js
├── test
│   └── metacoin.js
└── truffle.js
7 directories, 10 files
```

图 8.35 树形目录结构

该命令生成 4 个主目录，即 `app`、`contracts`、`migrations` 和 `test`。在前述示例中，共计生成了 7 个目录和 10 个文件，下面将对其逐一解释。

- ❑ `app`: 该目录涵盖了全部应用程序文件，包括 HTML 文件、图像、样式表以及 JavaScript 文件。同时，该文件夹还包含了子文件夹，即 `images`、`javascripts` 和 `stylesheets`，其中包含了相关应用程序。
- ❑ `contracts`: 该目录包含了 Solidity 合约源代码，这也是移植过程中 Truffle 查找 Solidity 合约文件之处。
- ❑ `migration`: 该目录包含了全部部署脚本。
- ❑ `test`: 顾名思义，该目录包含了应用程序和合约的测试文件。

最后，Truffle 配置文件存储于 `truffle.js` 文件中，并创建于 `truffle init` 运行的、项目的根文件夹中。作为示例，读者首先可了解 Truffle 各种命令的使用方式，并发布 `MetaCoin`。稍后，还将讨论如何针对自定义项目使用 Truffle。

库和合约均可采用 Truffle 进行编译。对此，合约文件名称应等同于文件中的合约名称。例如，在之前创建的 `MetaCoin` 项目中，位于 `contracts` 目录下的 `MetaCoin.sol` 文件与该文件中的 `MetaCoin` 合约具有相同的文件名称。同样，该机制也适用于库文件，且应区分大小写。

具体而言，当前文件名为 `MetaCoin.sol`；该文件中的合约名称为：

```
contract MetaCoin {  
    mapping (address => uint) balances;
```

编译过程如下所示：

```
~/testdapp$ truffle compile  
Compiling ConvertLib.sol...  
Compiling MetaCoin.sol...  
Compiling Migrations.sol...  
Writing artifacts to ./build/contracts  
~/testdapp$
```

当编译操作成功结束后，全部对象将写入 build 目录中。最终，输出路径如下所示：

```
~/testdapp$ tree build/  
build/  
├── contracts  
│   ├── ConvertLib.sol.js  
│   ├── MetaCoin.sol.js  
│   └── Migrations.sol.js  
1 directory, 3 files
```

其中，build 目录自动生成，并包含了 contracts 子目录，其中涵盖了 3 个 JavaScript 文件。

移植（迁移）过程是指 Truffle 将合约部署至区块链上。该过程依赖于 migrations 目录下的有效文件。

对应过程如下所示：

```
~/testdapp$ cd migrations/  
~/testdapp/migrations$ ls -ltr  
-rw-rw-r-- 1 drequinox drequinox 124 Dec 12 12:57  
2_deploy_contracts.js  
-rw-rw-r-- 1 drequinox drequinox 72 Dec 12 12:57  
1_initial_migration.js  
~/testdapp/migrations$ cat 2_deploy_contracts.js  
module.exports = function(deployer)  
{  
    deployer.deploy(ConvertLib);  
    deployer.autolink();  
    deployer.deploy(MetaCoin);  
};  
drequinox@drequinox-OP7010:~/testdapp/migrations$ cat  
1_initial_migration.js  
module.exports = function(deployer)  
{
```



```

    deployer.deploy(Migrations);
};

```

此处存在两个文件，其中包含了对应代码用以自定义须部署的合约。

相应地，文件名须采用数字前缀，以记录全部移植行为；而文件名中的后缀可表示为任意描述性名称。首先，须调整 `truffle.js` 文件，以定向至相应的网络。`truffle.js` 文件包含了与应用程序构建和 `rpc` 相关的各种信息。此处，`geth` 已处于运行状态，可简单地被定向并使用现有的客户端，如下所示：

```

module.exports = {
  build: {
    "index.html": "index.html",
    "app.js": [
      "javascripts/app.js"
    ],
    "app.css": [
      "stylesheets/app.css"
    ],
    "images/": "images/"
  },
  rpc: {
    host: "localhost",
    port: 8001
  }
};

```

在上述文件中，须修改 `rpc` 并定向至相应的网络。`rpc` 调整完毕后（在当前示例中，`geth` 运行于 8001 端口，而非常见的 8545 端口），`Truffle` 可通过相关命令执行迁移操作。需要注意的是，挖掘过程在 `rpc` 指向的以太坊节点上运行；否则，合约将无法被挖掘。

下列命令可用于部署合约：

```
~/testdapp$ truffle migrate
```

该操作可能会显示错误消息。这表明，用于将合约部署到区块链帐户的 `truffle` 被锁定，因而需要对其解锁，如下所示：

```

Running migration: 1_initial_migration.js
Deploying Migrations...
Error encountered, bailing. Network state unknown. Review successful
transactions manually.
Error: account is locked
    at Object.InvalidResponse
(/usr/lib/node_modules/truffle/node_modules/ethers/utils/
node_modules/web3/lib/web3/errors.js:35:16)

```

```

    at /usr/lib/node_modules/truffle/node_modules/etherpudding/
node_modules/web3/lib/web3/requestmanager.js:86:36
    at exports.XMLHttpRequest.request.onreadystatechange
(/usr/lib/node_modules/truffle/node_modules/web3/lib/web3/httpprovider.js:1
14:13)
    at exports.XMLHttpRequest.dispatchEvent
(/usr/lib/node_modules/truffle/node_modules/xmlhttprequest/lib/XMLHttpReque
st.js:591:25)
    at setState
(/usr/lib/node_modules/truffle/node_modules/xmlhttprequest/lib/XMLHttpReque
st.js:610:14)
    at IncomingMessage.<anonymous>
(/usr/lib/node_modules/truffle/node_modules/xmlhttprequest/lib/XMLHttpReque
st.js:447:13)
    at emitNone (events.js:91:20)
    at IncomingMessage.emit (events.js:185:7)
    at endReadableNT (_stream_readable.js:974:12)
    at _combinedTickCallback (internal/process/next_tick.js:74:11)
    at process._tickDomainCallback (internal/process/next_tick.js:122:9)

```

在 geth JavaScript 控制台中，采用下列命令解锁合约。

对此，首先需要列出账户并查看全部账户，并选择须解锁的账户。默认状态下，Truffle 假定为 coinbase，随后可选择对应的账户，如下所示：

```

> personal.listAccounts
["0x76f11b383dbc3becf8c5d9309219878caae265c3",
"0xcce6450413ac80f9ee8bd97ca02b92c065d77abc"]

```

账户解锁可采用下列命令：

```

> personal.unlockAccount("0x76f11b383dbc3becf8c5d9309219878caae265c3")
Unlock account 0x76f11b383dbc3becf8c5d9309219878caae265c3
Passphrase:
true

```

账户解锁完毕后，可通过下列命令再次执行移植操作：

```
~/testdapp$ truffle migrate
```

需要注意的是，挖掘工作应从移植阶段开始直至结束。通过 migrations 目录中的相关文件，移植行为将执行多个步骤。在当前示例中，1_initial_migration.js 和 2_deploy_contracts.js 文件用于提供 Truffle 所需的各项移植步骤，如下所示：

```

Running migration: 1_initial_migration.js
Deploying Migrations...

```



```

Migrations: 0xf444cce0cee00cab4d04bcfc0005626b8b02add8
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying ConvertLib...
  ConvertLib: 0x2ba8a4a75a6b845bf482923cff29ecc98cd68d90
  Linking ConvertLib to MetaCoin
  Deploying MetaCoin...
  MetaCoin: 0x0be9c5de978fa927b93a5c4faab31312cea5704a
Saving successful migration to network...
Saving artifacts...
~/testdapp$

```

一旦命令成功运行，将返回命令提示符，并显示“saving artifacts”消息。在 geth JavaScript 控制台中，部署工作可通过下列几条命令进行验证。

```

> eth.getBalance("0x0be9c5de978fa927b93a5c4faab31312cea5704a")
0
> eth.getCode("0x0be9c5de978fa927b93a5c4faab31312cea5704a")
"0x606060405260e060020a60003504637bd703e8811461003457806390b98a111461005657
8063f8b2cb4f1461007d575b610000565b346100005761004460043561009f565b604080519
18252519081900360200190f35b3461000057610069600435602435610119565b6040805191
15158252519081900360200190f35b34610000576100446004356101b1565b6040805191825
2519081900360200190f35b6000732ba8a4a75a6b845bf482923cff29ecc98cd68d906396e4
ee3d6100c4846101b1565b60026000604051602001526040518360e060020a0281526004018
08381526020018281526020019250505060206040518083038186803b156100005760325a03
f415610000575050604051519150505b919050565b600160a060020a0333166000908152602
0819052604081205482901015610142575060006101ab565b600160a060020a033381166000
818152602081815260408083208054889003905593871680835291849020805487019055835
1868152935191937fddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df5
23b3ef929081900390910190a35060015b92915050565b600160a060020a038116600090815
2602081905260409020545b91905056"

```

注意，最新部署的合约地址，将被之前的 Truffle 移植命令输出结果所获取（即 MetaCoin:0x0be9c5de978fa927b93a5c4faab31312cea5704a）。

- ❑ 与合约交互。Truffle 也提供了控制台（命令行界面），并支持与合约的交互行为。此时，全部合约均已初始化完毕并等待在控制台使用。该控制台基于 REPL，包含了 Read、Evaluate 和 Print Loop 等功能。类似地，在 geth 客户端中，REPL 通过 JSRE（JavaScript 运行期环境）使用，并可利用下列命令进行访问：

```
~/testdapp$ truffle console
```

这将开启如图 8.36 所示的命令行界面。

```
drequinox@drequinox-OP7010:~/testdapp$ truffle console
truffle(default)> █
```

图 8.36 Truffle 控制台

在控制台中，可运行各种相关方法，并查询合约。通过相关命令以及制表符，可显示方法列表，如图 8.37 所示。

```
drequinox@drequinox-OP7010:~/testdapp$ truffle console
truffle(default)> MetaCoin.
MetaCoin.  __defineGetter__      MetaCoin.  __defineSetter__      MetaCoin.  __lookupGetter__      MetaCoin.  __lookupSetter__
MetaCoin.  __proto__             MetaCoin.  constructor          MetaCoin.  hasOwnProperty      MetaCoin.  isPrototypeOf
MetaCoin.  propertyIsEnumerable  MetaCoin.  toLocaleString       MetaCoin.  toString           MetaCoin.  valueOf

MetaCoin.  apply                 MetaCoin.  arguments            MetaCoin.  bind                MetaCoin.  call
MetaCoin.  caller               MetaCoin.  length              MetaCoin.  name

MetaCoin.  abi                  MetaCoin.  address              MetaCoin.  all_networks        MetaCoin.  at
MetaCoin.  binary               MetaCoin.  checkNetwork           MetaCoin.  class_defaults            MetaCoin.  contract_name
MetaCoin.  currentProvider      MetaCoin.  defaults                MetaCoin.  deployed                  MetaCoin.  events
MetaCoin.  extend               MetaCoin.  generated_with          MetaCoin.  link                    MetaCoin.  links
MetaCoin.  network_id           MetaCoin.  networks                MetaCoin.  new                      MetaCoin.  next_gen
MetaCoin.  prototype            MetaCoin.  setNetwork               MetaCoin.  setProvider                MetaCoin.  unlinked_binary
MetaCoin.  updated_at           MetaCoin.  web3
```

图 8.37 显示方法列表

除此之外，还可调用其他方法，进而与合约交互。例如，当获取合约地址时，下列方法将在 Truffle 控制台中被调用。

```
truffle(default)> MetaCoin.deployed().address
'0x0be9c5de978fa927b93a5c4faab31312cea5704a'
truffle(default)>
```

❑ 查询合约余额。该操作如下所示：

```
truffle(default)>
MetaCoin.deployed().getBalance.call(web3.eth.accounts[0])
{ [String: '8750'] s: 1, e: 3, c: [ 8750 ] }
```

输出结果将返回包含值 8750 的字符串。

❑ 余额转账。该操作如下所示：

```
truffle(default)>
MetaCoin.deployed().sendCoin("0xcce6450413ac80f9ee8bd97ca02b92c065d77a
bc", 50, {from: "0x76f11b383dbc3becf8c5d9309219878caae265c3"})
'0xb8969149fcfb54ec9beac31af1fc86c386f9aa42cb13d2eb9bf9469931986e0f'
```

这将返回交易的哈希值。成功后，目标账户余额增加 50。其中，目标账户表示为传递至 sendCoin 函数中的参数。

□ 目标账户余额。该操作如下所示：

```
truffle(default)>
MetaCoin.deployed().getBalance.call(web3.eth.accounts[1])
{ [String: '1250'] s: 1, e: 3, c: [ 1250 ] }
truffle(default)>
```

最后，使用`.exit`命令可退出 Truffle 控制台。

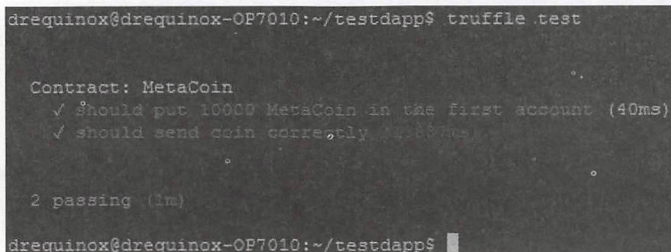
2. Truffle 测试

测试可视为 Truffle 的一项强大功能，并可通过下列命令调用：

```
~/testdapp$ truffle test
```

这将从`test`目录中读取、执行测试。Truffle 采用 Mocha 作为其测试框架，并利用 Chai 作为断言（assertion）框架。

该操作运行两项测试，如图 8.38 所示。在初始文件中，包含了 3 项测试，出于简单考虑，此处仅使用了其中的两项。另外，测试有可能在大多数平台中失败，因而须从文件中移除。稍后将对无效测试加以讨论。还需注意的是，挖掘应针对当前测试而运行。



```
drequinox@drequinox-OP7010:~/testdapp$ truffle test

Contract: MetaCoin
  ✓ should put 10000 MetaCoin in the first account (40ms)
  ✓ should send coin correctly (40ms)

2 passing (1m)
drequinox@drequinox-OP7010:~/testdapp$
```

图 8.38 Truffle 测试输出结果，其中包含了两项有效测试

其中，两项测试均在 Truffle 生成的文件的基础上进行。在下列文件中，出于简单考虑，此处仅显示了一项测试，此时针对默认的 MetaCoin 项目，Truffle 创建了 3 项测试。通过在文本编辑器中编辑`metacoin.js`文件，可从文件中移除测试。

```
contract('MetaCoin', function(accounts)
{
  it("should put 10000 MetaCoin in the first account",
  function()
  {
    var meta = MetaCoin.deployed();
    return meta.getBalance.call(accounts[0]).then(function(balance)
    {
```

```

    assert.equal(balance.valueOf(), 10000, "10000 wasn't in the first
account");
  });
});
});

```

全部测试文件须位于 `project` 文件夹中的 `tests` 文件夹下。同时，测试过程在 `it` 区块中进行，如图 8.39 所示。

```

drequinox@drequinox-OP7010:~/testdapp$ truffle test

Contract: MetaCoin
  ✓ should put 10000 MetaCoin in the first account

0
1 passing (1m)
drequinox@drequinox-OP7010:~/testdapp$

```

图 8.39 单项测试

该测试检测合约部署后余额是否为 10000。为进一步解释这一概念，可模拟某些错误行为。例如，如果 `metacoin.js` 文件：

```

assert.equal(balance.valueOf(), 10000, "10000 wasn't in the first
account");

```

被下列断言所修改：

```

assert.equal(balance.valueOf(), 1000, "10000 wasn't in the first
account");

```

则将导致人为的断言失败——在断言中，期望账户余额为 1000；而当合约部署后，该账户中包含了 10000 的余额。当运行测试时，如图 8.40 所示的输出结果表明，当前测试失败。需要说明的是，这一调整动作仅作为演示目的，以使用户查看测试失败状态，以及产生的输出类型。

```

drequinox@drequinox-OP7010:~/testdapp$ truffle test

Contract: MetaCoin
  ✓ should put 10000 MetaCoin in the first account
  > No events were emitted

0
0 passing (49s)

1) Contract: MetaCoin should put 10000 MetaCoin in the first account:
   AssertionError: 10000 MetaCoin in the first account (10000) is not equal to 1000
   at Test/metacoin.js:8:114
   at process._tickDomainCallback (internal/process/next_tick.js:129:7)

```

图 8.40 Truffle 测试失败时的输出结果

truffle test 命令接收若干个可选参数，特别是--verbose-rpc，该参数在理解以太坊客户端和 Truffle 之间的通信时特别有用。

在测试执行过程中，可能会产生如下错误消息。

```
Error: timeout of 120000ms exceeded. Ensure the done() callback is being called in this test.
```

若以太坊节点未进行挖掘，或者合约部署超出 2 分钟，则将会出现此类错误，这也体现了超时功能的作用。因此，对于 PrivateNet 网络，测试应在挖掘节点中执行；而在 Ropsten 中，时限则可以超出 2 分钟。除此之外，还可使用 ethereumjs-testrpc，并与 Truffle 结合使用，进而提供快速的以太坊 RPC 客户端。

3. build 命令

Truffle 中的 build 命令用于引导浏览器前端，同时须导入编译后的合约、所部署的相关合约以及以太坊客户端配置内容。在构建完毕后，全部对象均保存在 .build 目录中，而配置内容则位于 truffle.js 文件中，以对 Truffle 的构建内容进行引导。默认状态下，该文件仅包含 build:和 rpc:配置。

下列命令可启动 build。

```
~/testdapp$ truffle build
```

在构建过程结束后，将生成 build 目录（如果不存在），同时生成如下所示的树形结构。相应地，创建过程在 truffle.js 文件的基础上进行。

```
build/
├─ app.css
├─ app.js
├─ contracts
│  └─ ConvertLib.sol.js
│  └─ MetaCoin.sol.js
│  └─ Migrations.sol.js
├─ images
└─ index.html
```

此时，全部前端文件均已备齐，随后可使用 Truffle 的 serve 命令在浏览器中进行查看。此处，serve 命令构建一个 Web 服务器，以显示 HTML 内容。

需要注意的是，该命令须使用 -p 标记，以指定 TCP 7777 端口，如图 8.41 所示。其原因在于，geth 运行于 --rpccorsdomain 'http://localhost:7777'选项下，进而表明，仅支持 TCP 7777 上的内容。默认状态下，serve 将运行于 8080 端口，以供系统上的其他处理过程使

用。对于 Web 应用程序而言, TCP 8080 是一个较为常见的端口。

```
drequinox@drequinox-OP7010:~/testdapp$ truffle serve -p 7777
Serving app on port 7777...
Rebuilding...
Completed without errors on Mon Dec 12 2016 22:18:50 GMT+0000 (GMT)
```

图 8.41 Truffle 服务

一旦 Truffle 服务器启动并在相应的端口上运行, 通过浏览器并定向至 <http://localhost:7777>, 即可浏览相关内容, 如图 8.42 所示。

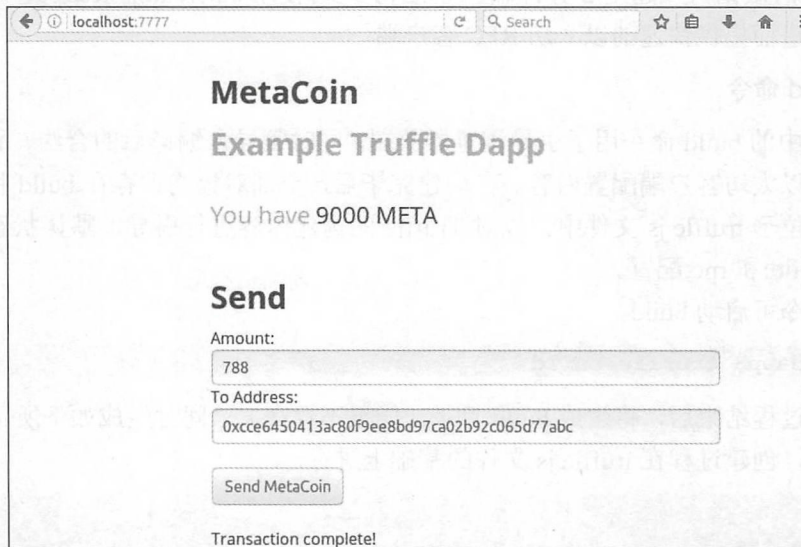


图 8.42 MetaCoin 前端示例

4. 示例

该示例通过移植和测试机制, 在 Solidity 中生成一个简单的合约。该合约较为简单, 且仅执行加法运算。

(1) 创建名为 simpleTest 的目录, 如下所示:

```
$ mkdir simpleTest
```

(2) 初始化 Truffle, 如下所示:

```
$ truffle init
```

(3) 从目录中移除文件。注意, 这一步骤不可或缺, 旨在移除 Truffle 创建的默认

MetaCoin 项目文件。

```
rm -r test/* contracts/* migrations/*
```

(4) 在合约目录中放置 Addition.sol 和 Migrations.sol 文件:

Addition.sol:

```
pragma solidity ^0.4.2;
contract Addition
{
    uint8 x;
    function addx(uint8 y, uint8 z )
    {
        x = y + z;
    }
    function retrievex() constant returns (uint8)
    {
        return x;
    }
}
```

Migrations.sol:

```
pragma solidity ^0.4.2;
contract Migrations
{
    address public owner;
    uint public last_completed_migration;
    modifier restricted()
    {
        if (msg.sender == owner) _;
    }
    function Migrations()
    {
        owner = msg.sender;
    }
    function setCompleted(uint completed) restricted
    {
        last_completed_migration = completed;
    }
    function upgrade(address new_address) restricted
    {
        Migrations upgraded = Migrations(new_address);
    }
}
```

```

    upgraded.setCompleted(last_completed_migration);
  }
}

```

(5) 在 test 目录中设置 Addition.js 文件:

```

contract('Addition', function(accounts)
{
  it(" 100 + 100 = 200 ", function()
  {
    var AddContract = Addition.deployed();
    AddContract.addx(100, 100, {from:accounts[0],gas:1000000})
    .then(function(a)
    {
      return AddContract.retrievex.call().then(function(Result)
      {
        assert.equal(Result, 200, "100 + 100 = 200 is expected");
      });
    });
  });
});

```

(6) 在 migrations 目录中设置两个文件。

□ `_initial_migration.js`:

```

module.exports = function(deployer)
{
  deployer.deploy(Migrations);
};

```

□ `_deploy_contracts.js`:

```

module.exports = function(deployer)
{
  deployer.deploy(Addition);
  deployer.autolink();
};

```

(7) 待全部文件就绪后, 使用 `truffle compile` 命令编译所有合约。作为可选项, 还可使用 `--compile-all` 标记重编译合约, 即使该合约已被编译过:

```

~/simpleTest$ truffle compile
Compiling Addition.sol...
Compiling Migrations.sol...
Writing artifacts to ./build/contracts

```

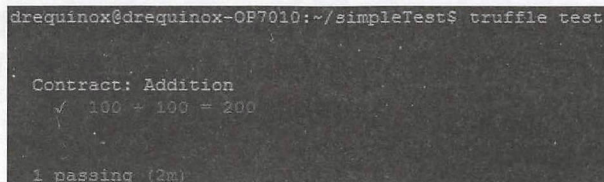

(8) 利用 `truffle migrate` 命令移植至以太坊测试网络中，这将在该网络上部署合约。注意，此时 `truffle.js` 文件须通过 8001 端口再次更新，以定向至 PrivateNet。

```
~/simpleTest$ truffle migrate
Running migration: 2_deploy_contracts.js
  Deploying Addition...
  Addition: 0x73934227a1ce7fc44152b7451626759a00b0275c
Saving successful migration to network...
Saving artifacts...
```

最后，可使用下列命令执行测试，此类测试基于前述 `Addition.js` 文件。

```
~/simpleTest$ truffle test
```

该命令在以太坊网络（当前为 PrivateNet）上部署合约，最终结果如图 8.43 所示。



```
dreguinox@dreguinox-OP7010:~/simpleTest$ truffle test

Contract: Addition
  ✓ 100 + 100 = 200

1 passing (2m)
```

图 8.43 Truffle 测试的示例输出结果

(9) 当与合约交互时，由于 `Addition` 合约已实现初始化，且已出现于 Truffle 控制台中，因而可通过各种方法与该合约进行交互。

例如，当获取部署合约的地址时，可调用下列方法。

```
truffle(default)> Addition.address
'0x73934227a1ce7fc44152b7451626759a00b0275c'
```

当从合约中调用函数时，可利用包含合约函数的部署方法。在下面的示例中，`addx` 函数被调用，并传递两个参数。

```
truffle(default)> Addition.deployed().addx(100,100)
'0xae6f51782c1bcf04ec34dd54ee31da626dc138993ea813bc6c3c1fe0790b130e'
truffle(default)>
'0xb9f8633fbd626466ee2c2f24952a5fca3134f4e7d08f39a4d26ac2689e22b653'
```

从当前合约中调用 `retrieveX` 函数，如下所示：

```
truffle(default)> Addition.deployed().retrieveX()
{ [String: '200'] s: 1, e: 2, c: [ 200 ] }
```

5. 项目示例：概念证明

该示例程序背后的理念是提供一种服务来公证文档，随后可用作某种证明，在过去某一段时间内，索赔人可以获得某些特定信息。这对专利文件非常有用。例如，如果某人提出了一个概念，可创建文档的哈希值并将其保存在区块链上。由于区块链的不可变性质，因而可将其作为永久证明——在某一特定时间存在的概念（文档）。对此，存在多种方法可实现这一方案，但关键思想基本相同，并遵循如下原理：哈希函数提供文本文摘或文档，且具有唯一性。

因此，关键思想是生成文档或文本的哈希值，并将其保存至区块链中。当文本执行了哈希计算并保存后，保存同一文本将被禁止，即在当前文档的哈希值和已存储的哈希值之间进行比较。

针对当前示例，需要使用到 Solidity、truffle 和 TestNet（已运行了之前创建的 Network ID 786）。首先，需要编写合约代码，可通过文本编辑器或集成开发环境予以实现。另外，浏览器 Solidity 也可用作模拟测试环境。通过该示例，读者可了解合约的开发流程，其中包括相关概念、Solidity 合约源代码以及最终的部署工作。

下面逐一考察各行代码，如下所示：

```
pragma solidity ^0.4.0;
```

该语句确保使用的最小编译器版本为 0.4.0，最大版本号不应超出 0.4.9，进而确保程序间的兼容性。

```
contract PatentIdea {
```

该语句表示合约的开始部分，该合约名为 PatentIdea。

随后可定义映射，并将 byte32 映射至布尔值，基本上是基于 byte32 的 hashmap（字典）映射至布尔值。

```
bool alreadyStored;
```

该变量利用 alreadyStored 名称声明，定义为布尔类型并可包含 true 或 false 值。以下变量用于加载源自 SaveIdeaHash 函数的返回值。

```
event ideahashed(bool);
```

事件被声明后，可用于捕捉哈希函数（SaveIdeaHash）。若事件被触发，将返回一个 true 或 false 布尔值。

下面声明 saveHash 函数，该函数接收 byte32 类型的哈希变量作为参数，进而导致合约状态发生变化。需要注意的是，作为合约内部函数，须将其修改为私有函数，如下所示：


```
function saveHash(bytes32 hash) private
{
    hashes[hash] = true;
}
```

`saveIdeaHash` 函数接收字符串变量作为参数，并根据函数结果返回布尔值（`true` 或 `false`），如下所示：

```
function SaveIdeaHash(string idea) returns (bool)
{
    var hashedIdea = HashtheIdea(idea);
    if (alreadyHashed(HashtheIdea(idea)))
    {
        alreadyStored=true;
        ideahashed(false);
        return alreadyStored;
    }
    saveHash(hashedIdea);
    ideahashed(true);
}
```

该函数包含一个变量 `hashedIdea`，在调用了 `HashtheIdea` 函数（稍后将对该函数加以解释）后将被赋值。注意，该函数在保存后将返回一个值，出于简单考虑，此处未予显示。

`alreadyHashed` 函数接收类型为 `byte32` 的哈希值，并在检测哈希映射中的哈希值后返回布尔值（`true` 或 `false`）。此处，该函数再次声明为常量函数，并定义为私有函数，如下所示：

```
function alreadyHashed(bytes32 hash) constant private returns(bool)
{
    return hashes[hash];
}
```

`isAlreadyHashed` 函数检测当前概念是否已哈希化，接收字符串类型参数并声明为常量函数。这也表明，合约的状态无法修改，并根据 `alreadyHashed` 函数的执行结果返回 `true` 或 `false`。随后，函数调用之前声明的 `alreadyHashed` 函数，并检测哈希映射（哈希值是否于此处被存储）。这意味着，同一字符串（`idea`）已被哈希化及存储，如下所示：

```
function isAlreadyHashed(string idea) constant returns (bool)
{
    var hashedIdea = HashtheIdea(idea);
    return alreadyHashed(hashedIdea);
}
```

最后，`HashtheIdea` 函数接收字符串类型的 `idea` 变量，并定义为 `constant` 函数，也就是说，合约的状态无法改变。鉴于该函数仅用于合约内部，因而其声明为 `private`。最后，函数将返回一个 `bytes32` 类型的数值。

```
function HashtheIdea(string idea) constant private returns (bytes32)
{
    return sha3(idea);
}
```

该函数调用 Solidity 中的内建函数 `sha3`，将字符串传递至 `idea` 变量中，最后返回一个字符串 `sha3` 哈希值。在 Solidity 中，`sha3` 函数为 `keccak256` 函数的别名，负责计算传递于其中的字符串 `Keccak-256` 哈希值。注意，这并非是 NIST 标准中的 `SHA-3`，而是定义为 `Keccak-256`，即 `SHA-3` 标准制定过程中 NIST 的最初提案，经适当调整后形成了 NIST 的 `SHA-3` 标准。相比于 `Keccak-256`（以太坊中的 `sha3` 函数），实际 `SHA-3` 标准哈希函数将返回不同的哈希结果。

完整的合约源代码如下所示：

```
pragma solidity ^0.4.0;
contract PatentIdea
{
    mapping (bytes32 => bool) private hashes;
    bool alreadyStored;
    event ideahashed(bool);
    function saveHash(bytes32 hash) private
    {
        hashes[hash] = true;
    }
    function SaveIdeaHash(string idea) returns (bool)
    {
        var hashedIdea = HashtheIdea(idea);
        if (alreadyHashed(HashtheIdea(idea)))
        {
            alreadyStored=true;
            ideahashed(false);
            return alreadyStored;
        }
        saveHash(hashedIdea);
        ideahashed(true);
    }
    function alreadyHashed(bytes32 hash) constant private returns (bool)
    {

```



```
return hashes[hash];
}
function isAlreadyHashed(string idea) constant returns (bool)
{
    var hashedIdea = HashtheIdea(idea);
    return alreadyHashed(hashedIdea);
}
function HashtheIdea(string idea) constant private returns (bytes32)
{
    return sha3(idea);
}
}
```

源代码可在浏览器 Solidity 中模拟执行，进而对其正确性予以验证。

一旦完成了合约源代码的输入和验证工作，对应结果将如图 8.44 右侧面板所示。

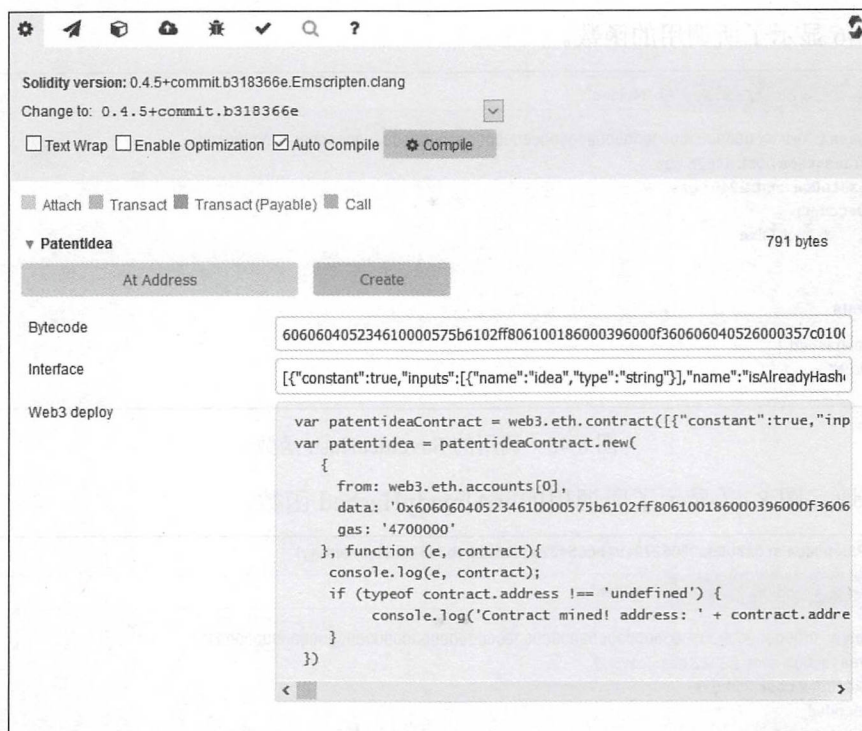


图 8.44 利用浏览器 Solidity 创建合约

上述代码仍存在较大的改进空间。例如，日期可存储于包含文档哈希值的某个映射中，并在查询时返回。另外，代码还可进行适当扩展，例如添加与专利相关的结构和信

息。出于简单性和易于理解方面的考虑，代码适当地降低了复杂度，进一步改善将留与读者以作练习。

单击 Create 按钮将显示合约中的两个函数，如图 8.45 所示。

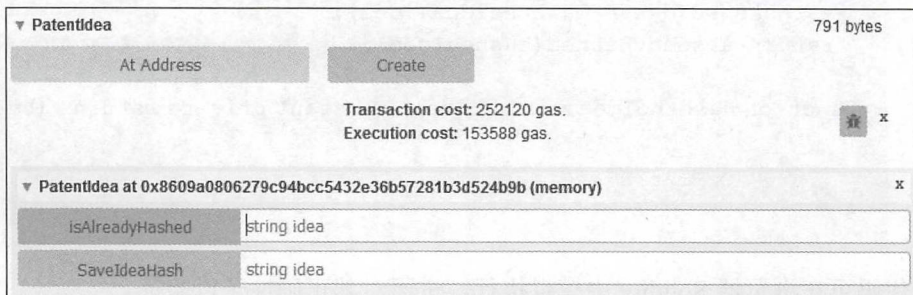


图 8.45 相关成本以及对应的两个函数

图 8.46 显示了所调用的函数。

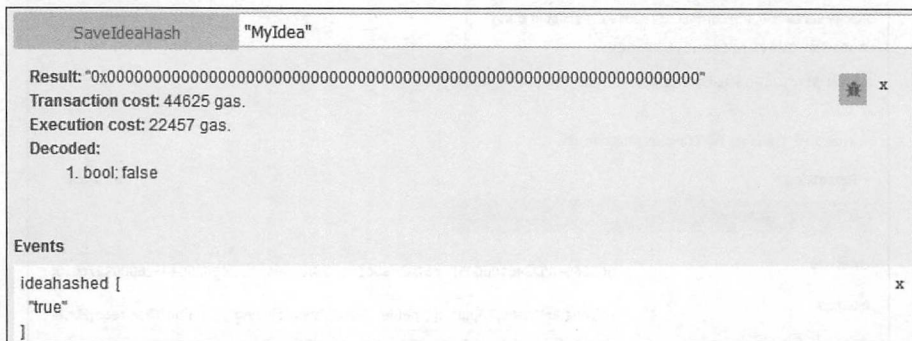


图 8.46 调用的 SaveIdeaHash 函数

类似地，图 8.47 显示了所调用的 isAlreadyHashed 函数。

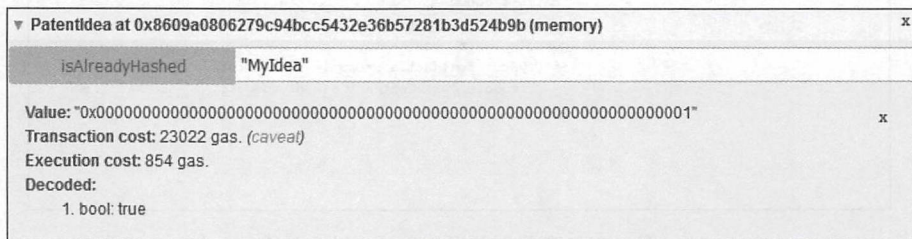


图 8.47 执行 isAlreadyHashed 函数

如果同一字符串再次传递至当前函数，将不会被保存，如图 8.48 所示。

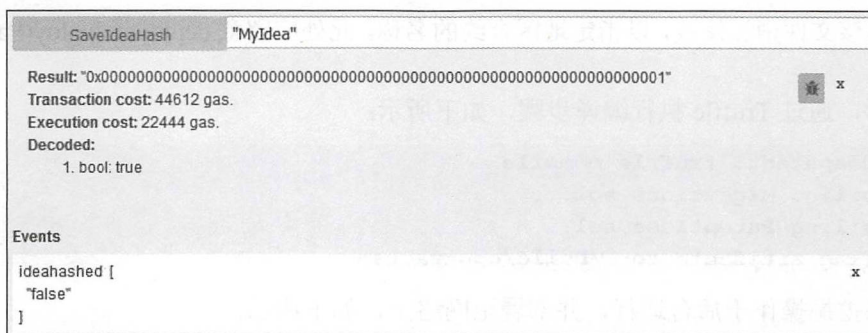


图 8.48 执行 SaveIdeaHash 函数

除此之外，还需注意当前事件返回了 false 值，表明哈希值无法保存且函数返回 true，进一步强调已经保存了相同的哈希值。

当在浏览器 Solidity 中模拟合约时，下一个步骤是使用 Truffle 初始化新项目，随后在 PrivateNet (ID786) 上部署、测试该合约，前述内容已对此有所介绍。

首先需要针对当前项目生成独立的目录，如下所示：

```
~$ mkdir ideapatent
~$ cd ideapatent/
```

随后，初始化 Truffle 并创建新项目，如下所示：

```
~/ideapatent$ truffle init
```

当项目创建完毕后，移除样本合约，如下所示：

```
~/ideapatent/contracts$ rm MetaCoin.sol ConvertLib.sol
```

在 contracts 文件夹中，创建名为 PatentIdea.sol 的文件，并将源代码置于该文件中。编辑 truffle.js 文件，使其定向至本机 HTTP 端点，如下所示：

```
rpc:
{
  host: "localhost",
  port: 8001
}
```

在 ~/ideapatent/migrations 文件夹下，编辑 2_deploy_contracts.js 文件，如下所示：

```
module.exports = function(deployer)
{
  deployer.deploy(PatentIdea);
  deployer.autolink();
};
```

可对该文件稍加修改，以指定地区合约的名称。此处应留意 `deployer.deploy(PatentIdea)` 函数。

随后，通过 Truffle 执行编译步骤，如下所示：

```
~/ideapatent$ truffle compile
Compiling Migrations.sol...
Compiling PatentIdea.sol...
Writing artifacts to ./build/contracts
```

确保挖掘操作于后台运行，并部署至网络上，如下所示：

```
~/ideapatent$ truffle migrate
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0x34d63de23de9c9b48251cec94fff427b94976109
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying PatentIdea...
  PatentIdea: 0x515fd6a5dbc1eb609dc1700f73be040d9db50d4b
Saving successful migration to network...
Saving artifacts...
```

待合约发布完毕后，可通过 Truffle 与控制台进行交互。

对此，利用下列命令启动 Truffle，如下所示：

```
~/ideapatent$ truffle console
```

控制台启动并运行后，源自发布合约中的函数即可被调用。

例如，可注册一个新的 idea，如下所示：

```
truffle(default)> PatentIdea.deployed().SaveIdeaHash("MyIdea")
'0x8644dc66f1173a9103034e17b761f8871ab10ef2a7d19bec9c7eb7164272b8a3'
```

检查 MyIdea 是否已哈希化，如下所示：

```
truffle(default)> PatentIdea.deployed().isAlreadyHashed("MyIdea")
true
```

检查另一个 idea 是否已被哈希化，如下所示：

```
truffle(default)> PatentIdea.deployed().isAlreadyHashed("MyOtherIdea")
false
truffle(default)>
```

该示例展示了如何在私有网络上创建、模拟和部署合约。当在 TestNet (Ropsten) 或



区块链上部署合约时，也可采用类似的方式。简单地讲，可定向至相应的 RPC，并采用 Truffle 移植进而在所选区块链上进行部署。

6. Oracle

在第 6 章曾有所讨论，Oracle 可作为智能合约的数据源，并涵盖多种服务。除此之外，Oraclize 也是另一个值得关注的话题，读者可访问 <http://www.oraclize.it/> 下载 Oraclize。若智能合约需要使用到来自第三方的现行价格或真实数据（如某一特定城市的气象条件），Oraclize 将十分有用。当根据真实事件制定决策时，在智能合约的受信数据方面，Oracle 可提供大量用例。相比之下，Oraclize 可简化智能合约与互联网之间的访问，从而方便地获得所需数据。

当在以太坊上使用 Oraclize 时，交易须发送至 Oraclize 合约中，除此之外，还应辅以相关支付和查询操作。最终，Oraclize 将根据所请求的交易提供的查询操作获取相关结果，并于随后将其发送回合约地址。当交易传回至当前合约时，将调用回调函数或回退函数。

在 Solidity 实际操作过程中，首先需要导入 Oraclize 库，进而可使用继承自该库的全部方法。当前，Oraclize 仅可用于 PrivateNet（Ropsten）以及 Live Main Net 以太坊区块链上。

图 8.49 显示了 Oraclize 的处理过程。

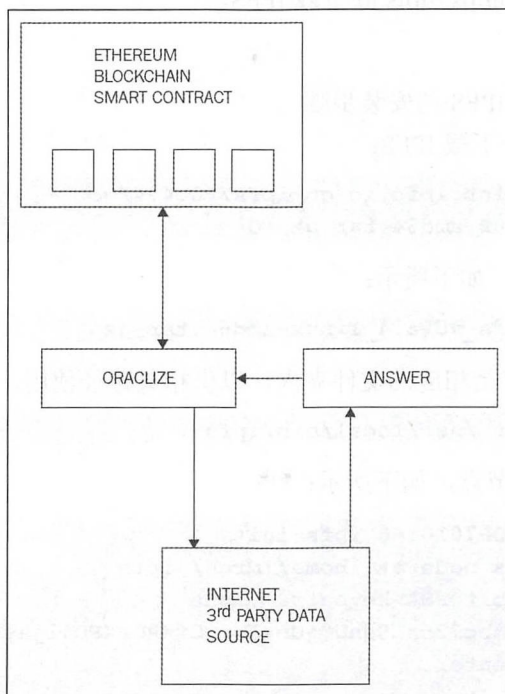


图 8.49 Oraclize 数据流



图 8.4 中显示了基于 Oraclize 的 Solidity 合约结构。需要注意的是，导入工作仅适用于 Web 所提供的开发环境，此类文件须通过手动方式导入，如下所示：

```
import "dev.oraclize.it/api.sol";
contract MyOracleContract is usingOraclize
{
    function MyOracleContract(){
    }
}
```

查询行为可采用如下类似方式：

```
oraclize_query("URL", "api.somewebsite.net/price?stock=XYZ");
```

另外，Oraclize 还可使用 TLS 公证，以确保输入内容的安全性和可靠性。

第 1 章曾提及，若希望从去中心化平台中获益，则需要对存储和通信层执行去中心化操作。传统上，Web 内容一般通过中心服务器提供，但通过分布式文件系统，该部分内容也可实现去中心化效果。

相应地，HTML 内容可存储于分布式、去中心化 IPFS 网络上，以实现增强型去中心化效果。

对此，读者可访问 <https://ipfs.io/> 下载 IPFS。

7. 安装 IPFS

下列各步骤显示了 IPFS 的安装步骤。

(1) 通过下列命令下载 IPFS：

```
$ curl https://dist.ipfs.io/go-ipfs/v0.4.4/go-
ipfs_v0.4.4_linux-amd64.tar.gz -O
```

(2) 解压 gz 文件，如下所示：

```
$ tar xvfz go-ipfs_v0.4.4_linux-amd64.tar.gz
```

(3) 将 ipfs 文件移至相应的文件夹内，以供相关路径使用，如下所示：

```
$ mv go-ipfs/ipfs /usr/local/bin/ipfs
```

(4) 初始化 IPFS 节点，如下所示：

```
imran@drequinox-OP7010:~$ ipfs init
initializing ipfs node at /home/imran/.ipfs
generating 2048-bit RSA keypair...done
peer identity: Qmbc726pLS9nUQjUbeJUxcCfXAGaXPD41jAszXniChJz62
to get started, enter:
    ipfs cat
/ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79oJWnPbdG/readme
```




(5) 输入图 8.50 中的命令，确保 IPFS 成功安装。

```
imran@drequinox-OP7010:~$ ipfs cat /ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG/readme
Hello and Welcome to IPFS!

IPFS

If you're seeing this, you have successfully installed
IPFS and are now interfacing with the ipfs merkledag!
```

图 8.50 IPFS 安装成功

(6) 启用 IPFS 后台程序，如下所示：

```
imran@drequinox-OP7010:~$ ipfs daemon
Initializing daemon...
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/192.168.0.17/tcp/4001
Swarm listening on /ip4/86.15.44.209/tcp/4001
Swarm listening on /ip4/86.15.44.209/tcp/41608
Swarm listening on /ip6:::1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
```

(7) 通过下列命令将文件复制至 IPFS 中：

```
~/sampleproject/build$ ipfs add --recursive --progress .
added QmVdYdYluycf32e8NhMVEWSufMyvcj17w3DkUt6BgeAtx7
build/app.css
added QmSypieNFeiUx6Sq7moAVCsgQhSY3Bh9ziwXJAxqSG5Pcp
build/app.js
added QmaJWMjD767GvuwuaLpt5tck9dTV CZPJ a9sDcr8vdcJ8pY
build/contracts/ConvertLib.sol.js
added QmQdz9eG2Qd5kwaU86kWebDGPqXBWj1Dmv9MN4BRzt2srf
build/contracts/MetaCoin.sol.js
added QmWpvBjXTP4HutEsYUh3JLDi8VYp73SKNji4aX1T6jwcmG
build/contracts/Migrations.sol.js
added QmQs7j6NpA1NMueTXKyswLaHKq3XDUCRay3VrC392Q4JDK
build/index.html
added QmPvWzyTEfLQnozDTfgdAAF4W9Bub2cDq5KUURpHrukseA
build/contracts
added QmUNLLsPACcz1vLxQVkJX5R1X345qqfHbsf67hvA3Nn
```



```
build/images
added QmSxpucr6J9rX3XQ3MBG8cVzLCrQFFKmMkTmpeNpjbtf3j build
```

(8) 当前，浏览器中的访问效果如图 8.51 所示。



图 8.51 通过 IPFS 访问 Web 页面



注意：

URL 定向至 IPFS 文件系统中。

(9) 最后，为了生成永久性变化，还需执行下列命令：

```
/build$ ipfs pin add QmSxpucr6J9rX3XQ3MBG8cVzLCrQFFKmMkTmpeNpjbtf3j
pinned QmSxpucr6J9rX3XQ3MBG8cVzLCrQFFKmMkTmpeNpjbtf3j recursively
```

上述示例展示了 IPFS 的应用方式，进而针对智能合约的 Web 部分（用户界面）提供去中心化结果。

IPFS 还可采用另一种方式予以使用。对于区块链而言，存储一直是一个较为重要的问题。对此，可存储大量的数据，同时将指向该数据的链接置于区块链交易中。通过这种方式，区块链中无须再存储大量数据，从而避免了数据膨胀问题。对此，可将数据置于 IPFS 上，并在区块链交易中存储 IPFS 链接，从而可引用所存储的数据。

以太坊自身的 Swarm 协议仍处于开发阶段，其工作原理并无太多变化。有鉴于此，目前 IPFS 仍不失为一种较好的选择。对于去中心化存储，IPFS 工作良好，并有望成为最终的选择平台。Swarm 支持轻量级客户端，并在其中存储全部区块链数据。Swarm 随 geth 当前版本推出，读者可访问 <https://swarm-guide.readthedocs.io/en/latest/introduction.html> 获取详细信息。考虑到成书时其仍处于开发阶段，因而具体信息尚不完备，相信这一状况很快会有所变化。

对于以太坊中的去中心化通信，Whisper 协议提供了相应的去中心化通信层，并可视作基于身份的消息层。另外，Web 3.0 技术也有望对 Swarm 和 Whisper 协议提供有力的支持。



8. 授权分布式账本

授权分布式账本的概念在根本上不同于公共区块链。分布式账本背后的关键思想是，其须经过适当授权许可，而不是公开的区块链。由于全部参与者均已通过网络审核，且无须执行挖掘操作即可确保网络安全，因而 DLT 不支持任何挖掘工作。同时，私有许可分布式账本上也不存在数字货币这一概念——经授权许可的区块链其目标不同于公共区块链。在公共区块链中，访问对每个人都是开放的，因而需要某种形式的激励和网络效应才能促使其成长；相反，在授权许可的 DLT 中则不存在此类要求。在私人机构背景下，特别是在现有的金融系统内工作，可以使用以太坊建立须授权许可的 DLT。分布式分类系统的优点在于其具备更快的速度以及可治理性，并且可与现有的金融系统进行交互操作。

8.5 本章小结

本章深入讨论了以太坊开发环境的配置方法以及智能合约的构建示例。首先介绍了私有以太坊网络的应用方法，以实现测试和开发用途。随后考察了 Solidity 语言，包括基础内容和语法知识。除此之外，还介绍了基于 geth 和 Web3 的部署方案，涉及了智能合约的开发和部署步骤，同时还结合实例探讨了各种开发技术。在讨论开发框架时，还辅以相关实例，以使读者更好地体验以太坊区块链中的智能合约的开发生命周期。本章篇幅较长，并涉及了大量的实例，读者可深度理解合约在以太坊上的开发、测试以及部署过程。最后，本章还介绍了与去中心化存储、去中心化通信以及 Oracle 相关的各种概念和工具。考虑到以太坊及其技术和框架均处于快速发展中，随着时间的推移，相信还会出现更加高级的工具和技术方案，但基本的框架性内容并不会出现太大的改动。限于篇幅，本章内容不可能面面俱到，但全部工具和技术均为当前主流应用，并为读者打下了坚实的基础；同时，读者也可轻松地过渡至更加高级的内容。后续章节中还将讨论智能合约的安全性、智能合约的形式验证、基于云服务的区块链以及不同领域内智能合约的特定用例。



第9章 超级账本

超级账本并非是区块链，而是一个由 Linux 基金会于 2015 年 12 月发起的项目，以推进区块链技术。该项目是由其成员共同努力，以构建一个开放源码的分布式账本框架，可以用来开发和实现跨行业的区块链应用程序和系统，其重心是构建和运行支持全球业务交易的平台。此外，该项目还致力于提高区块链系统的可靠性和性能。

超级账本项目经历了不同的发展阶段，包括提议、孵化和活跃期。当然，项目也可以被弃用，或者终止开发过程。在项目的孵化阶段，须配置一个完整的工作代码库，以及一个活跃的开发人员社区。

9.1 项目

目前，超级账本旗下包含 6 个项目，即 Fabric、Iroha、Sawtooth lake、Blockchain explorer、Fabric chaintool 和 Fabric SDK Py。其中，Corda 是最近添加的，预计有望成为超级账本项目中的一员。该项目目前拥有 100 名成员，以及 120 多位较为活跃的贡献者，他们定期举行会议，并在全球范围内组织会谈。

下面将对这些项目进行简要介绍，之后将提供更多关于 Fabric 和 Sawtooth lake 的设计、架构和实现细节。

9.1.1 Fabric

Fabric 是由 IBM 和 DAH（数字资产控股）提出的区块链项目，并为区块链解决方案的开发提供支持。Fabric 基于插接式架构，必要时，可以将各种组件（如共识算法）接入至系统中。读者可访问 <https://github.com/hyperledger/fabric> 以了解该项目。

9.1.2 Sawtooth lake

Sawtooth lake 是英特尔在 2016 年 4 月提出的区块链项目，其中涵盖了某些创新亮点，例如，将账本从交易中分离出来（解耦），在多个业务领域中灵活地使用交易族（transaction



families)，以及可插接的共识机制。

此处，解耦可解释为：交易从共识层分离出来，并使用交易族这一新概念。也就是说，不再采用单独地与账本结合在一起的交易事务，交易族支持更加灵活、丰富的语义，以及不受限制的业务逻辑设计。读者可访问 <https://github.com/hyperledger/sawtooth-core> 以了解该项目。

9.1.3 Iroha

Iroha 由 Soramitsu、日立、NTT Data 和 Colu 于 2016 年 9 月提出。Iroha 的目标是构建可复用的组件库，用户经选择后可在超级分布式账本上运行。通过提供 C++编写的可重用组件，Iroha 旨在实现其他超级账本项目，并重点关注移动开发。另外，该项目还提出了一种新的共识算法，称为 Sumeragi，该算法是一种基于链式的拜占庭容错共识算法。关于 Iroha，读者可访问 <https://github.com/hyperledger/iroha> 获取更多信息。Iroha 包括但不限于数字签名库（ed25519）、SHA-3 哈希库、交易序列化库、P2P 库、API 服务器库、iOS 库、Android 库和 JavaScript 库。

9.1.4 Blockchain explorer

Blockchain explorer 项目计划为 Hyperledger 构建一个区块链浏览器，用于查询来自区块链的交易、区块和相关数据。除此之外，还提供了网络信息和与链代码交互的能力。此外，目前还有另外两个项目正在酝酿之中：Fabric chaintool 和 Fabric SDK Py。这一类项目旨在对 Hyperledger Fabric 提供支持。

9.1.5 Fabric 链式工具

超级账本链式码编译器用于支持 Fabric 链式码开发，其目的是构建一个在高级别谷歌协议缓冲结构中读取并生成链式码的工具。此外，该编译器还可将链式码打包，以便直接部署。预计该工具可在开发阶段提供各种帮助，例如编译、测试、打包和部署。读者可访问 <https://github.com/hyperledger/fabric-chaintool> 获取更多信息。

9.1.6 Fabric SDK Py

该项目将构建一个基于 Python 的 SDK 库，可以用来与区块链（Fabric）进行交互。



读者可访问 <https://github.com/hyperledger/fabric-sdk-py> 获取更多信息。

9.1.7 Corda

作为最新项目，Corda 是由 R3 提供给 Hyperledger 项目的，并于 2016 年 11 月 30 日实现了开源。Corda 主要面向金融服务行业，并与金融行业的主要银行和组织合作开发。在本书编写时，Corda 尚未完全纳入 Hyperledger 项目下。从技术上讲，Corda 并不是区块链，但其关键特性与区块链相似，例如共识、有效性、唯一性、不变性和身份验证。

下面将对 Fabric (IBM)、Sawtooth lake (Intel) 以及 Corda (R3) 分别加以讨论。

9.2 超级账本协议

超级账本旨在建立一个新的区块链平台，并由行业用例加以驱动。对此，社区对超级账本项目做出了突出的贡献，超级账本区块链平台逐渐形成一种商业交易协议。超级账本正在演变成为一种规范，并可作为构建区块链平台的一类参考方案。与以前的区块链解决方案相比，超级账本只针对特定类型的行业或需求。后续章节将介绍由 Hyperledger 项目发布的参考架构。由于这项工作仍处于持续发展中，因此最终结果可能会产生某些变化，但核心服务应不会有太大变化。

9.2.1 参考架构

Hyperledger 发表了一份白皮书，其参考架构可以作为构建授权分布式账本的指导原则。参考体系结构包括两个主要组件：Hyperledger 服务和 Hyperledger API、SDK 和 CLI。Hyperledger 服务包含了身份验证服务、策略服务、区块链服务和智能合约服务等。另一方面，Hyperledger API、SDKs 和 CLIs 通过适当的应用程序编程接口、软件开发工具包或命令行接口提供了一个面向区块链服务的接口。此外，事件流基本上是一个 gRPC 通道且贯穿所有服务，并可以接收和发送事件。这里，事件并非是预先定义或自定义的。相应地，验证对等点或链式码将产生外部应用程序响应或侦听的事件。

在本书编写时，Hyperledger 白皮书中发布的参考架构如图 9.1 所示。需要注意的是，Hyperledger 项目正处于发展中，此处所展示的架构预计会有所改变。

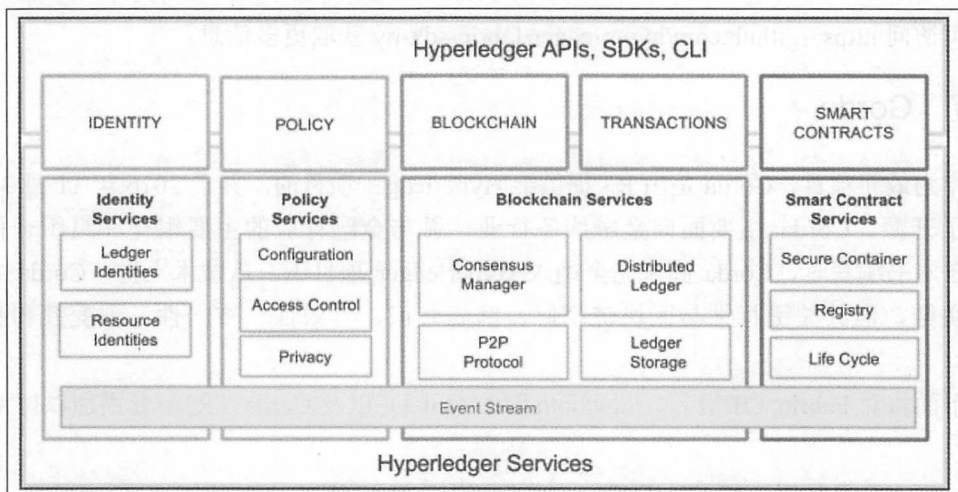


图 9.1 Hyperledger 白皮书草案（最新版本为 V2.0.0）中的
Hyperledger 架构（数据源自 Hyperledger 白皮书）

9.2.2 需求条件

区块链服务存在一定的要求。具体而言，参考架构是由 Hyperledger 项目的参与者，以及研究行业用例的需求所驱动的。从工业用例的研究中可归纳出一些需求类别，稍后将对此加以讨论。

模块化结构是 Hyperledger 核心需求。可以想象，作为一种交叉技术（区块链），模块化结构可用于多种业务场景。因此，与存储、策略、链式码、访问控制、共识和许多其他区块链服务相关的功能都应该具有插接式特征。也就是说，模块应该是“即插即用”型，用户可轻松地删除和添加满足业务需求的不同模块。

例如，如果业务区块链仅在受信方之间运行，并执行基本的业务操作，通常不需要使用到高级的加密技术，以支持机密和隐私性。因此，用户应可移除该项功能（模块），或者使用更为适宜的模块以满足具体需求。同样，如果用户需要运行一个跨行业的区块链，那么机密性和隐私可能是最重要的。在这种情况下，用户应该能够将高级加密和访问控制机制（模块）接入至区块链中。

9.2.3 隐私和保密性

在业务区块链中，交易和合约的隐私和机密性是至关重要的。因此，超级账本旨在

提供广泛的加密协议和算法，用户可根据其业务需求选择适当的模块。该结构应该能够处理复杂的加密算法，且不会对性能产生影响。

9.2.4 身份

为了提供隐私和机密性服务，须定义灵活的 PKI 模型，可用于处理访问控制功能。加密机制的强度和类型也将根据用户的需求而变化。在特定场景中，用户可能需要隐藏其身份，超级账本应支持此类功能。

9.2.5 可审核性

可审核性是 Hyperledger Fabric 的另一项要求，对于所有身份、相关操作和更改，应保持其审计跟踪（audit trail）的不变性。

9.2.6 互操作性

目前存在多种区块链解决方案，但彼此间无法相互通信。针对基于区块链的全球商业生态系统增长，这也是一个限制因素。许多区块链网络将在业务领域中针对特定需求进行操作，但重要的是，区块链间应能够相互通信。对此，应该制定所有区块链须遵循的一类通用标准，以便在不同的账本之间进行通信，并最终形成一项协议，允许在多个 Fabrics 之间交换信息。

9.2.7 可移植性

可移植性与跨平台和多种环境运行下的能力有关，且无须在代码级别上更改任何内容。Hyperledger 的设计原则包含了可移植性，涉及基础设施层面、代码、库和 API，进而支持跨平台实现的一致性开发原则。

9.3 Fabric

在了解 Hyperledger 孵化项目之前，首先应考察 Hyperledger 中的某些基础内容以及相关术语。下面首先讨论 Fabric 这一概念。

Fabric 可以定义为一个组件集合，提供了一个可以用来交付区块链网络的基础层。Fabric 网络有各种类型和功能，但是所有的 Fabric 都具有相同的属性，例如不可变性和

共识驱动。某些 Fabric 可以为构建区块链网络提供模块化方法。在这种情况下，区块链网络可设置多个可插接模块，并在网络上执行各种功能。例如，共识算法可表示为区块链网络中的可插接模块。根据网络的具体需求，可以选择相应的共识算法并将其接入至网络。这一类模块可以基于某些特定的 Fabric 规范，涉及 API、访问控制和其他各种组件。另外，Fabric 也可以设计为私有或公有类型，并可创建多个业务网络。例如，作为一类应用程序，比特币运行在其 Fabric（区块链网络）之上。如前所述，区块链可以是经过授权许可的，也可以是未经任何授权的，这同样适用于 Hyperledger 中的 Fabric。

另外，Fabric 这一名称，也体现了 IBM 对 Hyperledger 基金会所做出的代码贡献，其正式名称为 Hyperledger Fabric。同时，IBM 还通过 Bluemix 云服务提供了区块链服务（IBM 区块链）。

9.4 Hyperledger Fabric

Fabric 是 IBM 最初为 Hyperledger 项目所做出的贡献成果，旨在通过模块化、开放和灵活的方案构建区块链网络。Fabric 中的各种功能具有“可插拔”特征，同时支持任何语言开发智能合约——Fabric 基于容器技术，并可以承载任何语言。链式码（智能合约）被沙箱包装成一个安全的容器，其中涵盖了安全的操作系统、链式码语言、运行时环境和 SDK（针对 Go、Java 和 node.js）。必要时，还可以采用其他语言。智能合约在 Fabric 中称为链式码。与以太坊中的特定语言，或者比特币中非常有限的脚本语言相比，这是一个非常强大的特性。它是一个被许可的网络，旨在解决诸如可伸缩性、隐私和机密性等问题。这背后的关键思想是模块化技术，进而支持灵活的设计和实现方案，并满足可伸缩性、隐私和其他所需的属性。对于一般用户来说，Fabric 中的交易具有私有性、机密性和匿名性，但仍可被授权的审计人员跟踪并链接到用户。作为授权许可网络，所有参与者都须在成员服务中注册，以访问区块链网络。另外，该账本还提供了可审核功能，以满足各项规章制度。

9.4.1 Fabric 体系结构

根据所提供的服务类型，Fabric 通过逻辑方式划分为 3 个主要类别，包括成员服务、区块链服务和链式码服务。下面将详细讨论所有类别和相关组件。目前，较为稳定的 Hyperledger Fabric 版本是 v0.6；而最新版本 v1.0 尚缺乏稳定性，且出现了某些架构上的变更。在本章后面的部分中，将讨论 1.0 版本中的变化内容。

1. 会员服务

针对 Fabric 网络的用户，这一类服务用于提供访问控制功能，下列内容显示了会员服务的功能：

- ❑ 用户身份验证。
- ❑ 用户注册。
- ❑ 根据具体角色，为用户分配适当的权限。

会员服务利用公钥基础框架（PKI）支持身份管理和授权操作。会员服务由多种组件构成，如下所示：

- ❑ 注册授权（RA）。对用户进行身份验证，并评估 Fabric 参与者的身份。
- ❑ 注册证书授权。注册证书（电子证书）是由 ECA 向注册参与者颁发的长期证书，目的是为参与该网络的实体提供身份证明。
- ❑ 交易证书授权。为了在网络上发送交易，参与者须持有交易证书。对此，TCA 负责向注册证书持有者颁发交易证书，并由上述电子证书派生而来。
- ❑ TLS 证书授权。为了保证 Fabric 上节点之间的网络级通信，可采用 TLS 证书。TLS 证书授权机构颁发 TLS 证书，以确保此类消息在区块链网络上不同系统之间进行传递。

2. 区块链服务

区块链服务是 Hyperledger Fabric 的核心内容，其组件如下所示：

- ❑ 共识管理器。共识管理器负责为共识算法提供接口。作为一个适配器，接收来自其他 Hyperledger 实体的交易，并根据所选算法类型在既定标准下执行。这里，共识具有可插接特征，目前，Fabric 中存在 3 种共识算法，即批处理 PBFT 协议、SIEVE 算法和 NOOPS。
- ❑ 分布式账本。区块链和全局（世界）状态是分布式账本的两个主要元素。其中，区块链简单地表示为一个链接的区块列表（前述章节曾对此有所介绍），而全局账本则是一个键-值数据库，数据库用于智能合约在交易执行期间存储相关状态。区块链由包含交易的区块组成。其中，交易包含了链式码，执行相关交易并更新全局状态。每个节点在 RocksDB 中保存了磁盘上的全局状态。图 9.2 显示了 Hyperledger Fabric 中的区块，其中包含了相关字段。

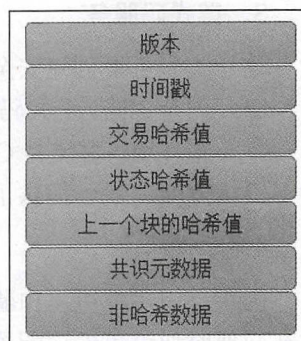


图 9.2 区块结构

图 9.2 中各字段解释如下：“版本”用于记录协议的变化；“时间戳”表示 UTC 时间戳，并由区块申请者更新；“交易哈希值”包含了区块中交易的 Merkle 根节点哈希值；“状态哈希值”表示世界状态的 Merkle 根节点哈希值；“上一个块的哈希值”表示前一区块的哈希值并在区块消息序列化之后计算，随后将生成基于 SHA3 SHAKE256 算法的消息文摘；作为可选字段，“共识元数据”可用于共识协议中，进而提供某些与当前共识相关的信息；“非哈希数据”表示存储于区块中的元数据，且尚未哈希化。这一特性使得不同对等节点上可包含不同的数据。除此之外，还可弃用数据且不会对区块链产生任何影响。

- ❑ P2P 协议。采用谷歌 RPC (gRPC) 构建 Hyperledger Fabric 中的 P2P 协议，同时利用协议缓冲区定义消息结构。

消息在节点之间传递，进而执行各种功能。Hyperledger Fabric 中包含 4 种主要消息类型：Discovery、交易、同步和共识。其中，Discovery 消息在节点之间交换，以发现网络上的其他对等点。

交易消息可分为两种类型：部署交易和调用交易。前者用于将新的链式码部署到账本上；后者用于从智能合约中调用函数。交易可定义为公共交易、保密交易以及保密链式码交易。其中，公共交易具有开放特性，所有参与者均可使用；而保密交易仅允许交易所有者和参与者查询。保密链式码交易设置了加密的链式码，仅可通过验证节点解密。相应地，验证节点执行共识机制、验证交易并维护区块链。另外一方面，非验证节点则提供交易验证、流服务器和 REST 服务，同时还作为交易者和验证节点之间的代理。最后，同步消息由对等点调用，以使区块链更新状态，并与其他节点同步。共识消息则用于共识管理和载荷广播中，进而验证对等点。这些都是由共识框架于内部生成的。

- ❑ 账本存储。RocksDB 用于保存账本状态，并将其存储在每个对等点中。RocksDB 是一个高性能的数据库，读者可访问 <http://rocksdb.org/> 以了解更多信息。

3. 链式码服务

这一类服务可创建安全容器，用于执行链式码，分类如下：

- ❑ 安全容器。链式码部署于 Docker 容器中，为智能合约的执行提供了一个锁定的沙箱环境，并支持 Golang 这一类智能合约语言。必要时，也可采用其他主流语言。
- ❑ 安全注册。提供了包含智能合约的全部图像记录。

4. 事件

区块链中事件可以由验证器节点和智能合约触发。外部应用程序可以侦听此类事件，并在必要时通过事件适配器对其予以响应。此处的事件机制类似于第 8 章 Solidity 中引入的事件概念。

5. API 和 CLI

应用程序编程接口通过公开各种 REST API 为 Fabric 提供了相关接口。此外，还提供了 REST API 子集的命令接口，并支持快速测试以及与区块链之间的有限交互。

9.4.2 Fabric 组件

各种组件均可构成区块链的一部分内容，包括但不限于账本、链式码、共识机制、访问控制、事件、系统监视和管理、钱包和系统集成组件。

1. 对等点或节点

Fabric 网络上可执行两种类型的对等点，即验证型和非验证型。简单地说，验证节点执行共识机制，创建并验证事务，并有助于更新账本和维护链式码。

相比之下，非验证对等点不执行任何交易，仅构造那些随后被转发到验证节点的交易。两种类型的节点管理和维护由成员服务发布的用户证书。

2. 区块链上的应用程序

Fabric 上的典型应用程序仅由一个用户接口构成，通常采用 JavaScript/HTML 编写，并与后端链式码（智能合约）交互。此处，后端链式码通过 API 层存储于账本中，如图 9.3 所示。

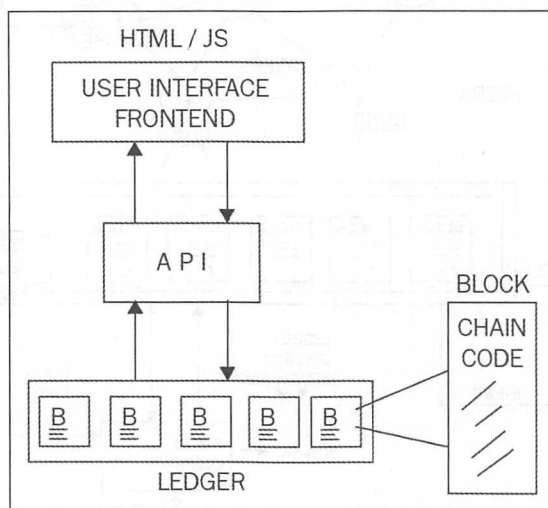


图 9.3 典型的链式码应用程序

Hyperledger 提供各种 API 和命令行接口，以支持与账本的交互。对应 API 包括身份、交易、链式码、账本、网络、存储和事件的接口。

链式码通常是在 Golang 或 Java 中编写的，一般包括公共型、机密型或访问控制型。这一类代码作为智能合同，用户可以通过 API 与之进行交互。另外，用户可以调用链式码中的函数，这将导致状态更改，进而更新账本。除此之外，一些函数只用于查询分类账，不会导致任何状态更改。

当实现链式码时，首先须在代码中创建链式码 shim 接口，这可以通过 Java 代码或 Golang 代码予以实现。针对于此，需要使用到以下 4 个函数：

(1) Init()函数。当链式码部署于账本中时，需要调用该函数，这将初始化链式码并导致状态变化，进而更新账本。

(2) Query()函数。该函数用于查询部署链式码的当前状态。注意，该函数并不改变当前账本的状态。

(3) Main()函数。当对等点部署其链式码副本时，将执行该函数。利用基于该函数的对等点即可注册链式码。

图 9.4 显示了 Hyperledger Fabric 的整体结构。

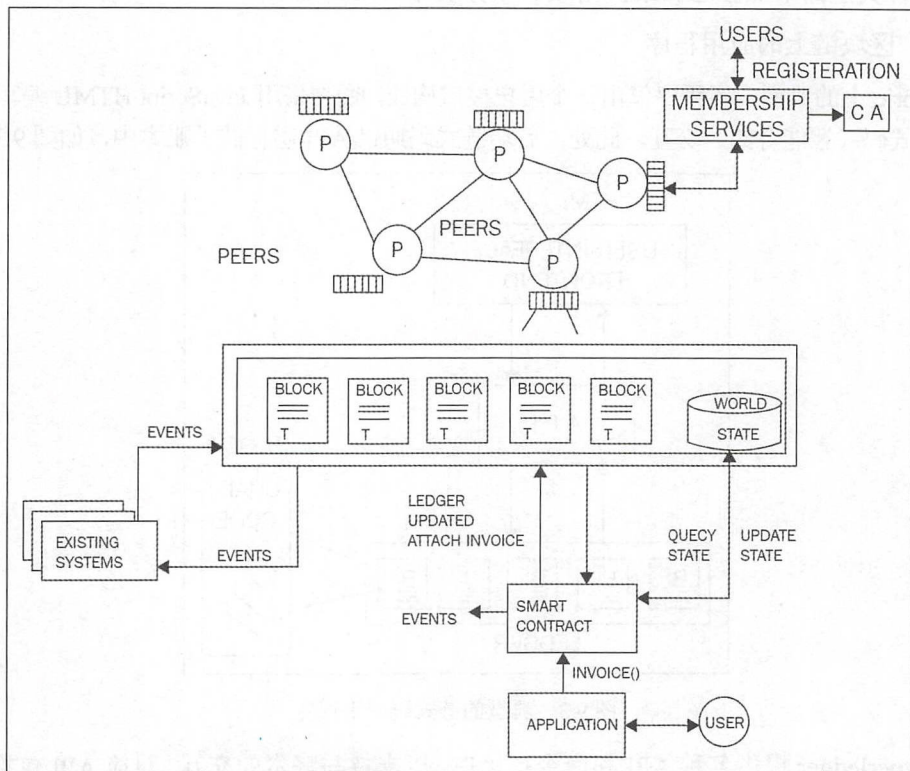


图 9.4 Hyperledger Fabric 的高层结构

Hyperledger Fabric 区块链应用程序遵循 MVC-B 架构，基于流行的 MVC 设计模式。该模型中的组件包括模型、视图、控制和区块链。

- ❑ 视图逻辑。与用户界面相关，可以是桌面、Web 应用程序或移动前端。

- ❑ 控制逻辑。表示为用户界面、数据模型和 API 之间的协调器。

- ❑ 数据模型。该模型用于管理区块链之外的数据。

- ❑ 区块链逻辑。通过基于控制器的区块链和基于交易的数据模型管理区块链。

Hyperledger 的当前版本为 v0.6，且版本 v1.0 尚在构建之中，因而本节并不打算介绍相应的操作示例。

在本书出版时，Hyperledger 的设置信息有可能已经过时了。因此，读者应时刻关注其更新状态，对应网址为 <https://hyperledgerfabric.readthedocs.io/en/latest/>。

另外，IBM Bluemix 服务在其区块链服务下提供了区块链示例应用程序。对此，读者可访问 https://console.ng.bluemix.net/docs/services/blockchain/ibmblockchain_tutorials.html，该服务允许用户在易于使用的环境中创建自己的区块链网络。

9.5 Sawtooth lake

Sawtooth lake 既可以在授权许可模式下运行，也可以在非许可模式下运行。作为一种分布式账本，Sawtooth lake 提出了两种新的概念，引入了一种新的共识算法，即消逝时间量证明（PoET），以及交易族概念。下面将对此予以简要描述。

9.5.1 PoET

PoET 是一种新型共识算法，并根据节点的等待时间（在形成区块之前）随机选择一个节点。这与选举算法和基于工作量证明的彩票算法形成了鲜明的对比。在这些算法中，区块选择将消耗大量的电力和计算机资源，例如比特币。PoET 是一种工作量证明算法，但采用受信的计算模型提供一种满足工作量证明的机制，而不是花费计算机资源。PoET 利用英特尔的 SGX 架构提供一种受信的执行环境，以确保过程的随机性和密码安全性。需要注意的是，目前实施的 Sawtooth lake 并不需要真正的硬件 SGX（基于 TEE），且仅出于试验目的予以模拟，因此不应在产品环境中使用。

9.5.2 交易族

传统的智能合约范例提供了一种解决方案，且基于全域的通用指令集。例如，在以

以太坊案例中，已针对以太坊虚拟机（EVM）开发了一组操作码，可用于构建智能合约，以满足不同行业、不同类型的需求。虽然该模型优点较为突出，但缺陷也很明显，且缺乏应有的安全性——仅向账本提供了单一接口，有可能为恶意代码提供较大的攻击界面。这种复杂和通用的虚拟机范式涵盖了可被黑客发现和利用的漏洞。最近的例子是 DAO 攻击和拒绝服务（DoS）攻击，这一类攻击利用了某些 EVM 操作码的限制。图 9.5 所示模型描述了传统的智能合约模型，其中，通用虚拟机用于向全域区块链提供接口。

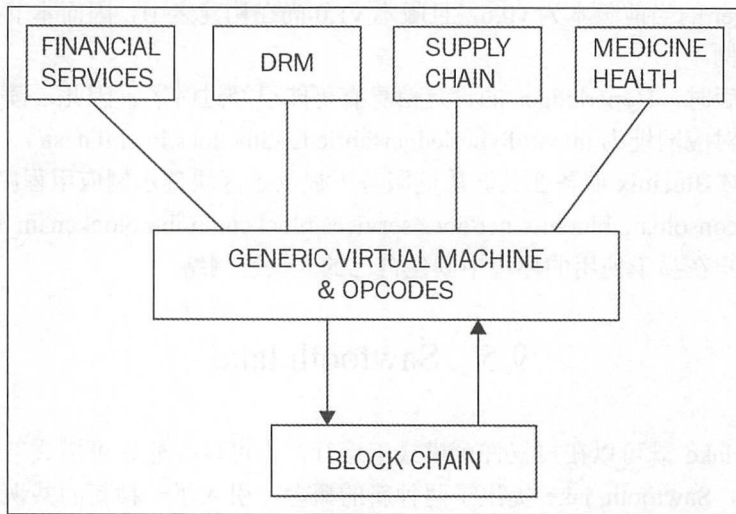


图 9.5 传统智能合约范例

为了解决这一问题，Sawtooth lake 提出了交易族的概念。交易族是通过将逻辑层分解成规则集和一个特定域的合成层来创建的。此处的关键思想是，业务逻辑于交易族中构成，提供了一种更安全、更强大的方式来构建智能合约。同时，交易族还包含了特定领域的规则，以及支持为该域创建交易的另一个层。从另一种角度来看，交易族是数据模型的组合，同时也是一种针对特定领域实现逻辑层的交易语言。数据模型体现了区块链（账本）的当前状态，而交易语言则负责修改账本的状态。用户将根据业务需求构建自己的交易族。

图 9.6 展示了上述模型，其中每个特定领域，如金融服务、数字版权管理（DRM）、供应链和医疗行业，均拥有自己的逻辑层（由特定于该领域的操作和服务组成）。因此，这使得逻辑层具备了一定的限制性且兼具强大的功能。交易族确保在控制逻辑中只存在与所需领域相关的操作，从而降低了负面操作的概率。

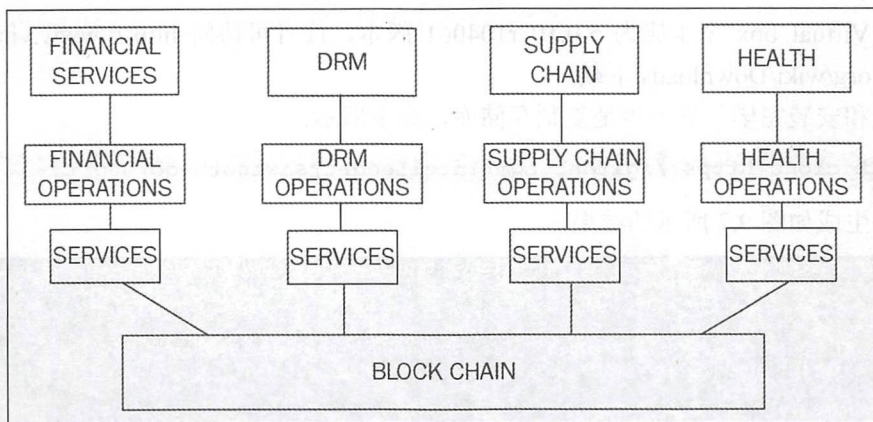


图 9.6 Sawtooth（交易族）智能合约范例

英特尔已经提供了 3 种包含 Sawtooth 的交易族，即 Endpoint registry、Integerkey 和 Marketplace。

- ❑ Endpoint registry 用于注册账本服务。
- ❑ Integerkey 用于测试已部署的账本。
- ❑ Marketplace 用于操作和服务之间的交易。

作为概念证明，Sawtooth_bond 已构成了一个债券交易平台。关于 Sawtooth_bond，读者可访问 <https://github.com/hyperledger/sawtooth-core/tree/master/extensions/bond> 以了解更多信息。

9.5.3 Sawtooth 中的共识机制

基于所选取的网络，Sawtooth 包含两种类型的共识机制。如前所述，PoET 可视为一个基于彩票功能的受信执行环境，它根据节点的等待时间（形成区块）随机选出一个领导者。除此之外，还存在另一种共识类型，称为仲裁投票，以适应 Ripple 和 Stellar 形成的共识协议。该共识算法支持即时交易结果，因而适用于授权许可的网络。

9.5.4 开发环境

本节将简要介绍 Sawtooth lake 的开发环境。为了设置开发环境，某些先决条件不可或缺。本节示例均假定运行于 Ubuntu 系统中，同时包含以下内容：

- ❑ vagrant 至少应为 1.9.0 版本，读者可访问 <https://www.vagrantup.com/downloads.html> 下载。

- ❑ Virtual box 至少应为 5.0.10 r104061 版本, 读者可访问 <https://www.virtualbox.org/wiki/Downloads> 下载。

下载和安装完毕, 下一步是复制存储库, 如下所示:

```
$ git clone https://github.com/IntelLedger/sawtooth-core.git
```

这将生成如图 9.7 所示的结果。

```
drequinox@drequinox-0P7010:~/project$ git clone https://github.com/IntelLedger/sawtooth-core.git
Cloning into 'sawtooth-core'...
remote: Counting objects: 12527, done.
remote: Compressing objects: 100% (964/964), done.
remote: Total 12527 (delta 452), reused 0 (delta 0), pack-reused 11515
Receiving objects: 100% (12527/12527), 9.26 MiB | 1.76 MiB/s, done.
Resolving deltas: 100% (8131/8131), done.
Checking connectivity... done.
```

图 9.7 复制 GitHub Sawtooth

Virtual box 复制完毕后, 下一步就是启动当前环境。首先, 运行以下命令将目录更改为正确的位置, 然后启动 vagrant box。

```
$ cd sawtooth-core/tools
$ vagrant up
```

这将产生如图 9.8 所示的结果。

```
drequinox@drequinox-0P7010:~/project/sawtooth-core/tools$ vagrant up
Could not determine vagrant user.
VAGRANT_BOX = ubuntu/xenial64
VAGRANT_FORWARD_PORTS = true
VAGRANT_MEMORY = 2048
VAGRANT_CPUS = 2
Proxyconf plugin not found
Install: vagrant plugin install vagrant-proxyconf
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Box 'ubuntu/xenial64' could not be found. Attempting to find and install...
default: Box Provider: virtualbox
default: Box Version: >= 0
==> default: Loading metadata for box 'ubuntu/xenial64'
default: URL: https://atlas.hashicorp.com/ubuntu/xenial64
==> default: Adding box 'ubuntu/xenial64' (v20161221.0.0) for provider: virtualbox
default: Downloading: https://atlas.hashicorp.com/ubuntu/boxes/xenial64/versions/20161221.0.0/providers/virtualbox.box
x default: Progress: 1% (Rate: 1709k/s, Estimated time remaining: 0:04:04)
```

图 9.8 Vagrant box 启动命令

终止 Vagrant box 则可采用下列命令:

```
$ vagrant halt
```

或

```
$ vagrant destroy
```

其中, halt 将终止 vagrant, 而 destroy 命令则终止并删除 vagrant。

最后, 交易验证器可通过下列命令启动, 首先需要在 vagrant Sawtooth box 执行 ssh 命令, 如下所示:

```
$ vagrant ssh
```

当出现 vagrant 提示符时, 运行后续各项命令。

首先, 通过下列命令构建 sawtooth lake 核心:

```
$ /project/sawtooth-core/bin/build_all
```

构建完毕后, 运行下列命令启动交易验证器:

```
$ /project/sawtooth-core/docs/source/tutorial/genesis.sh
```

这将生成创始区块并清除现有的数据文件和密钥, 最终结果如图 9.9 所示。

```
ubuntu@ubuntu-xenial:/project/sawtooth-core$ /project/sawtooth-core/docs/source/tutorial/genesis.sh
writing file: /home/ubuntu/sawtooth/keys/base000.wif
writing file: /home/ubuntu/sawtooth/keys/base000.addr
```

图 9.9 创始区块并生成密钥

运行交易验证器并按照下列方式调整路径:

```
$ cd /project/saw-toothcore
```

运行交易验证器, 如下所示:

```
$ ./bin/txnvalidator -v -F ledger.transaction.integer_key --config
/home/ubuntu/sawtooth/v0.json
```

对应结果如图 9.10 所示。

```
ubuntu@ubuntu-xenial:/project/sawtooth-core$ ./bin/txnvalidator -v -F ledger.transaction.integer_key --config /home/ubuntu/sawtooth/v0.json
22:08:22 INFO validator_cli validator started with arguments: ['./bin/txnvalidator', '-v', '-F', 'ledger.transaction.integer_key', '--config', '/home/ubuntu/sawtooth/v0.json']
22:08:22 INFO validator_cli read signing key from /home/ubuntu/sawtooth/keys/base000.wif
22:08:24 WARNING validator_cli validator pid is 10937
22:08:24 INFO gossip_core listening on IPv4Address(UDP, '0.0.0.0', 33713)
22:08:24 INFO global_store_manager create blockstore from file /home/ubuntu/sawtooth/data/base000_state.dbm with flag c
22:08:24 INFO validator set administration node to None
22:08:24 INFO validator starting ledger base000 with id 1K5RnedZ at network address ('127.0.0.1', 33713)
22:08:24 INFO web_api listen for HTTP requests on (ip='localhost', port=8800)
22:08:24 INFO validator_cli adding transaction family: ledger.transaction.integer_key
22:08:24 INFO journal_core restore ledger state from persistence
22:08:24 INFO global_store_manager add block 60af3ec894fa1cb0 to the queue for loading
22:08:24 INFO global_store_manager load block 60af3ec894fa1cb0 from storage
22:08:24 INFO journal_core commit head: 60af3ec894fa1cb0
22:08:26 INFO validator ledger connections using RandomWalk topology
22:08:26 INFO random_walk initiate random walk topology update
22:08:29 INFO validator ledger initialization complete
22:08:29 INFO journal_core process initial transactions and blocks
22:08:29 INFO validator register endpoint 1K5RnedZ with name base000
22:08:29 INFO journal_core build transaction block to extend 60af3ec8 with 1 transactions
22:08:29 INFO wait_timer wait timer created: TIMER, 5.00, 33.69, HE2DQNJWGI2DCNJQ
```

图 9.10 运行交易验证器

按下 Ctrl+C 快捷键可终止验证器节点。当验证器启动并运行时, 即可在另一个终端

窗口中启用各种客户端，进而与交易验证器进行通信并提交各项交易。

例如，图 9.11 启动了 market 客户端，并与交易验证器通信。需要注意的是，下列命令在 /keys/mkt.wif 下生成密钥：

```
./bin/sawtooth keygen --key-dir validator/keys mkt
```

该示例源自 Sawtooth lake 文档中的基本示例，实际开发过程较为复杂，甚至可独立成章。

```
ubuntu@ubuntu-xenial:/project/sawtooth-core$ ./bin/mktclient --name market --keyfile validator/keys/mkt.wif
//UNKNOWN> help
Documented commands (type help <topic>):
=====
EOF      dump      exit      liability  selloffer  tokenstore
account  echo      help      map        session    waitforcommit
asset    exchange  holding   offers     sleep
assettype exchangeoffer holdings participant state

Miscellaneous help topics:
=====
symbols  names

//UNKNOWN> participant reg --name market --description "the market"
transaction ff652e63dadeaf32 submitted
//market> █
```

图 9.11 交易族的 market 客户端

与此同时，Sawtooth lake 也处于不断发展中，读者应留意相关文档（对应网址为 <http://intelledger.github.io/>）以了解其最新进展。

9.6 Corda

Corda 并非区块链。如前所述，传统的区块链解决方案是将交易内容封装至一个区块中；同时，每个区块都以加密方式连接回其父区块，并提供了一种不可变的交易记录。然而，Corda 的情况则并非如此：Corda 经重新规划后采纳了区块链各种优点，但并未形成传统的区块链。Corda 完全是面向金融行业而发展起来的，因为每个组织管理自己的账本，同时也制定了自身的各项规则，这也带来了一定的矛盾和操作风险。此外，在各家组织机构之间，数据也会被复制，从而导致管理成本的增加（涉及各自的基础设施和复杂度）。通过建立一个去中心化的数据库平台，Corda 旨在解决金融行业所面临的问题。

读者可访问 <https://github.com/corda/corda> 下载 Corda 源代码，代码采用 Kotlin 语言编写，这是一种定位于 Java 虚拟机（JVM）的静态类型语言。

9.6.1 体系结构

Corda 平台的主要组件包括状态对象、合约代码、法律条文、交易、共识和工作流。下面主要介绍状态对象、交易、共识和工作流。

1. 状态对象

状态对象代表金融协定中最小的数据单位，并作为交易的执行结果而创建或删除，主要是指合约代码和法律条文。作为可选项，法律条文对合约具有法律约束力。然而，合约代码则具有强制性，并以此管理对象的状态。根据合约代码中定义的业务逻辑，这将为节点提供一种状态转换机制。状态对象包含表示对象当前状态的数据结构。例如，图 9.12 中，状态对象体现了对象的当前状态。在这种情况下，可视为 Party A 和 Party B 之间简单的模拟协议；在此基础上，Party ABC 已支付给 Party XYZ 1000 个 GBP。该过程表现了对对象的当前状态；然而，被引用的合约代码可以通过交易改变状态。这里，状态对象可以视作一个状态机，并被交易消耗以创建更新的状态对象。

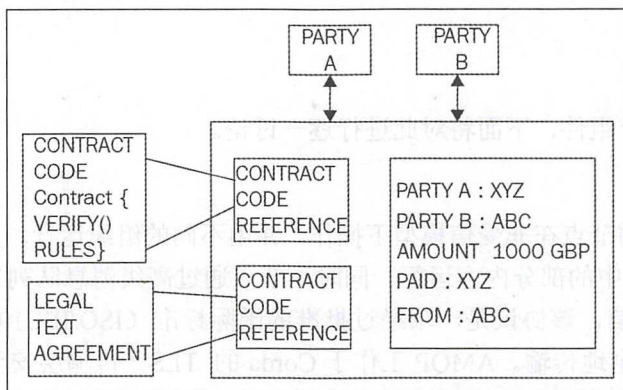


图 9.12 状态对象示例

2. 交易

交易用于在不同状态之间执行转换。例如，图 9.12 中显示的状态对象即作为交易结果而创建。另外，Corda 根据其交易处理模型使用了一个比特币式的 UTXO。交易的状态转换概念和比特币相同。与比特币类似，交易可能包含单个或多个输入（或者不包含任何输入内容），也可能涉及单个或多个输出。所有交易均经过数字签名。此外，Corda 并未定义挖掘这一概念——Corda 不使用区块在区块链中安排交易。相反，公证服务用于提供交易的时间顺序。在 Corda 中，可以使用 JVM 字节码开发新的交易类型，因而功能强大且兼具灵活性。

3. 共识

Corda 中的共识模型非常简单且基于公证服务（稍后将对此加以讨论）。一般的想法是，交易通过公证服务评估其唯一性。若具有唯一性，即被签署为有效。Corda 网络上可执行单个或多个集群的公证服务。公证机构可使用各种共识算法以达成某种共识，如 PBFT 或 Raft。

Corda 共识涉及两个主要的概念：状态有效性共识，以及状态唯一性共识。第一个概念涉及到交易的验证行为，以确保全部所需签名均有效且状态适宜。第二个概念是一种检测双重支付攻击的有效方法，确保交易未被消费且具有唯一性。

4. 工作流

Corda 中的工作流是一种新颖的思想，支持开发去中心化的工作流。Corda 网络上的所有通信均通过这一类工作流处理。作为交易构建协议，可通过代码定义包含任意复杂度的金融流程。工作流可作为异步状态机运行，并与其他节点和用户交互。在执行过程中，可以根据需要暂停或恢复自身状态。

9.6.2 组件

Corda 包含多个组件，下面将对此进行逐一讨论。

1. 节点

Corda 网络中的节点在非受信模型下操作，并由不同的组织运行。另外，节点作为身份验证的 P2P 网络中的部分内容运行。同时，节点通过高级消息队列协议（AMQP）实现彼此间的直接通信，该协议是一项经过批准的国际标准（ISO/IEC19464），确保跨不同节点的消息可安全地传输。AMQP 工作于 Corda 的 TLS（传输层安全）上，从而保证节点之间通信数据的隐私性和完整性。

节点还利用本地关系数据库进行存储。网络上的消息以紧凑的二进制格式编码，并通过 Apache Artemis 消息代理（Active MQ）交付和管理。相应地，节点还可以作为网络映射服务、公证、Oracle 或普通节点。图 9.13 显示了两个节点之间通信的高级视图。

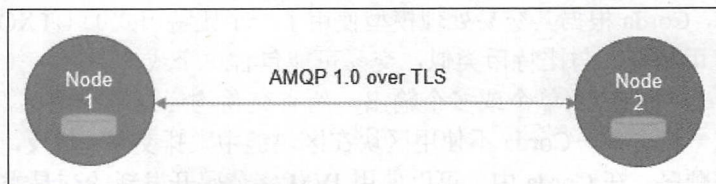


图 9.13 Corda 网络中两个节点间的通信

其中，Node 1 利用 AMQP 协议在 TLS 通信通道上与 Node 2 进行通信，节点采用了本地关系型数据库进行存储。

2. 授权服务

许可服务用于提供安全性 TLS 证书。对于网络参与行为，参与者应具备由根证书颁发机构颁发的签名标识。网络中的身份应具有唯一性，而授权许可服务则用于签署此类身份。另外，用于识别参与者的命名规则基于 X.500 标准，从而确保名称的唯一性。

3. 网络映射服务

该服务以网络所有节点的文档形式提供网络映射，发布 IP 地址、身份证书和节点提供的服务列表。所有节点在第一次启动时向该服务注册，以显示其存在。当节点接收到连接请求时，首先在网络映射上检查请求节点的存在。换句话说，该服务将参与者的身份解析为物理节点。

4. 公证服务

在传统的区块链中，挖掘用于确定包含交易的区块顺序；而在 Corda 中，公证服务用于提供交易顺序和时间戳服务。对此，网络中可以存在多个公证服务，并由组合公钥予以识别。根据应用程序的具体要求，公证服务可采用不同的共识算法，例如 BFT 或 Raft。公证服务将对交易进行签署，以表明交易的有效性和最终结果，随后可将其持久化至数据库中。

由于性能原因，公证服务可在负载均衡的配置环境中运行，以便将负载分散到节点上；而且，为了减少延迟，节点应在交易参与者附近运行。

5. Oracle 服务

Oracle 服务对包含真实结果的服务进行签署，或者自身提供真实数据；同时，还可将真实数据输入至分布式账本中。

6. 交易

Corda 网络中的交易并未采用全局方式予以传输，而是在半私有网络中传输，且仅在与交易相关的参与者的子集之间共享。这与传统的区块链解决方案形成了鲜明的对比，例如以太坊和比特币。其中，所有的交易以全局方式在网络上传播。同时，交易采用数字签名，使用相关状态或者创建新的状态。

Corda 网络中的交易由下列元素组成：

- ❑ 输入引用。表示交易所用的状态引用，并作为输入使用。
- ❑ 输出状态。交易生成的新状态。

- ❑ 附件。表示附加 Zip 文件的哈希列表。Zip 文件可以包含与交易相关的代码和其他相关文档。注意，文件本身并不是交易的一部分；相反，此类文件是单独传输和存储的。
- ❑ 命令。作为合约的参数，命令表示与预期交易操作相关的信息。其中，每个条命令包含一个公钥列表，代表所有需要签署交易的各方。
- ❑ 签名。表示交易所需的签名。所需签名的总数与命令的公钥数量成正比。
- ❑ 类型。交易一般存在两种类型：正常交易或公证变更。公证变更针对某一状态重新分配公证者。
- ❑ 时间戳。该字段表示交易发生的时间范围，并通过公证服务验证并执行。此外，时间要求较为严格的场合（例如金融服务），公证服务应该与原子钟同步。
- ❑ 摘要。表示交易操作的文本描述。

7. Vaults

Vaults 在某一个节点上运行，且类似于比特币的钱包概念。由于此类交易并非以全局方式传播，因此每个节点仅在其 Vaults 中包含部分关联数据。Vaults 将其数据存储在标准的关系型数据库中，因此可以使用标准的 SQL 语句查询数据。另外，Vaults 可以同时包含账本和非账本数据，也就是说可以包含某些账本之外的数据。

8. CorDapp

Corda 的核心模型由状态对象、交易和交易协议组成，该协议与合约代码、API、钱包插件和用户界面组件结合使用时，将构建一个 Corda 分布式应用程序（CorDapp）。

Corda 智能合约采用 Kotlin 或 Java 语言编写，其代码定位于 JVM，稍作调整后即可获得 JVM 字节码执行后的确定结果。Corda 智能合约中涵盖了 3 种主要组成部分：

- ❑ 可执行代码，定义了验证逻辑以验证对状态对象的更改。
- ❑ 状态对象表示合约的当前状态，可以供交易使用，也可以通过交易创建。
- ❑ 命令用于描述操作及验证数据，该数据定义了交易的验证方式。

9.6.3 开发环境

Corda 的开发环境并不复杂，下面针对相关步骤加以考察。

在执行开发任务之前，须安装以下软件：

- ❑ JDK 8, 读者可访问 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 下载。
- ❑ IntelliJ IDEA 社区版本，读者可访问 <https://www.jetbrains.com/idea/download> 免

费下载。

- ❑ H2 数据库平台, 读者可访问 <http://www.h2database.com/html/download.html> 下载。
- ❑ Git, 读者可访问 <https://git-scm.com/downloads> 下载。
- ❑ Kotlin 语言。关于该语言, 读者可访问 <https://kotlinlang.org/> 以了解更多信息。
- ❑ Gradle 则是构建 Corda 的另一个组件。

当全部工具安装完毕后, 即可开始智能合约的开发之旅。其中, CorDapps 开发可通过示例模板完成, 对应网址为 <https://github.com/corda/cordapp-template>; 另外, 读者可访问 <https://docs.corda.net/>, 并下载与智能合约开发相关的详细文档。

通过下列命令, 可从 GitHub 中实现 Corda 的本地复制。

```
$ git clone https://github.com/corda/corda.git
```

当复制操作完成后, 输出结果如下所示:

```
Cloning into 'corda'...
remote: Counting objects: 74695, done.
remote: Compressing objects: 100% (67/67), done.
remote: Total 74695 (delta 17), reused 0 (delta 0), pack-reused 74591
Receiving objects: 100% (74695/74695), 51.27 MiB | 1.72 MiB/s, done.
Resolving deltas: 100% (42863/42863), done.
Checking connectivity... done.
```

一旦存储库被复制, 就可以在 IntelliJ 中启用以供进一步开发使用。存储库中包含了多个可用的示例, 如 Corda 银行、利率交换、演示内容等。读者可以在 Corda 下的/samples 目录下找到它们, 并通过 IntelliJ IDEA IDE 予以查看。

9.7 本章小结

本章介绍了 Hyperledger 项目, 首先讨论了该项目背后的核心思想, 并简要介绍了相应的孵化项目。本章详细讨论了 3 种主要的 Hyperledger 项目, 即 Hyperledger fabric、Sawtooth lake 和 Corda, 且均处于开发过程中, 预计在下一个版本中会有所变化。因此, 本章并未提供相应的实践操作。尽管如此, 项目的核心内容基本保持不变, 读者可访问本章提供的相关链接, 以查看最新的更新结果。Hyperledger 项目在区块链技术中扮演着关键角色。本章中讨论的每个项目旨在解决不同行业所面临的各种问题, 并突破现有区块链技术中的某些局限性, 例如可伸缩性和隐私。预计不久的将来, Hyperledger 项目中的内容还将不断丰富, 通过多方协作以及开源精神, 区块链技术将获取极大的进步, 并将使整个社区受益。

第 10 章 替代区块链方案

本章主要介绍替代区块链的解决方案。随着比特币的成功以及区块链技术的实现，各种区块链协议、应用程序和平台的开发层出不穷。一些项目并没有获得太多的关注，但多数项目已在这一领域打下了坚实的基础。

本章将引入替代区块链和平台方案，它们是自身的新区块链，或者是现有区块链的补充内容。此类新平台基于 SDK 或相关工具，进而简化区块链的开发和部署。以太坊和比特币的成功促使各种各样的项目产生，并借助于各种基础技术和概念，从而发展壮大。这一类新项目的价值主要体现在：借助于用户友好的附加工具层，可解决当前区块链中的局限性问题或完善现有区块链内容。

本章首先介绍新的区块链解决方案，随后探讨与现有区块链互补的各种平台和开发工具包。例如，BlockApps STRATO 是一个与以太坊兼容的平台，用于开发区块链应用程序；而 Kadena 则是一个新出现的私有区块链，具有某些较新的特征，例如 Scalable BFT。在区块链技术的发展中，也首次引入了诸如侧链、驱动链和锚定（pegging）等概念。本章将详细讨论这一类技术和相关概念。当然，限于篇幅，本章不可能覆盖全部替代链和平台；但所有平台会随着区块链的引入而被介绍。

10.1 区 块 链

本节将介绍新的区块链解决方案。下面首先讨论名为 Kadena 的新区块链。

Kadena 是最近出现的私有区块链，成功解决了区块链系统中的可伸缩性和隐私问题。另外，一种名为 Pact 的非图灵完备语言也随着 Kadena 而被引入，并支持智能合约的开发。Kadena 的一个关键创新是其 Scalable BFT 共识算法，并在保持性能的前提下扩展到数千个节点。这里，Scalable BFT 基于原始 Raft 算法，同时也是 Tangaroa 和 Juno 的继承者。Tangaroa，其名称源自包含容错机制的 Raft 实现（BFT Raft），以解决 Raft 算法中拜占庭节点行为的可用性和安全性问题；而 Juno 则是 JPMorgan 开发的一分支（分叉）。第 1 章曾详细讨论了共识算法。这两项提议都存在一个基本的限制条件——无法在维持高性能的同时进行扩展。因此，Juno 并未获得巨大成功。随着节点数量的增加，私有区块链在维护高性能方面具有更理想的特性。Kadena 用其专有的 Scalable BFT 算法解决了这个

问题, 该算法预计将扩展到数千个节点, 且不存在任何性能下降。

此外, 保密性也是 Kadena 的另一个重要话题, 以在区块链上进行隐私交易。这是通过使用密钥转置、对称链加密、增量哈希和 Double Ratchet 协议这一组合加以实现的。

密钥转置可视作保证私有区块链安全性的标准机制。如果密钥被破坏, 通过周期性地改变加密密钥, 则可阻止任何攻击。智能合约语言 Pact 针对密钥转置提供了本地支持。

对称链加密支持区块链上加密交易数据, 此类交易可被特定私有交易的参与者自动解密; 而 Double Ratchet 协议用于提供密钥管理和加密功能。

Scalable BFT 共识协议确保在智能合约执行之前完成复制并达成共识。达成共识的过程如下所示:

- ❑ 首先, 新出现的交易经用户签名后在区块链网络上进行广播, 该过程由领导者节点选取, 并将其添加至不可变日志中。此时, 还将针对该日志计算增量哈希值。其中, 增量哈希表示为一种哈希函数类型, 并支持哈希消息计算。如果之前已哈希化的原始消息稍有改变, 则新哈希消息将根据现有哈希值进行计算。与传统的哈希函数相比, 当前方案具有更快的计算速度, 且占用更少的资源。对于前者, 即使原始消息只发生了较小的变动, 也需要生成一个全新的哈希消息。
- ❑ 一旦交易被领导节点写入到日志中, 即对复制结果和增量哈希予以标记, 并将其广播到其他节点中。
- ❑ 其他节点在接收交易后, 将验证领导者节点的签名, 并将交易添加到自己的日志中, 同时向其他节点广播自己的计算增量哈希 (法定证明)。最后, 在从其他节点接收到足够数量的证明之后, 交易将永久地提交至账本中。

上述过程的简化版本如图 10.1 所示, 其中领导者节点负责记录新交易, 随后将其复制

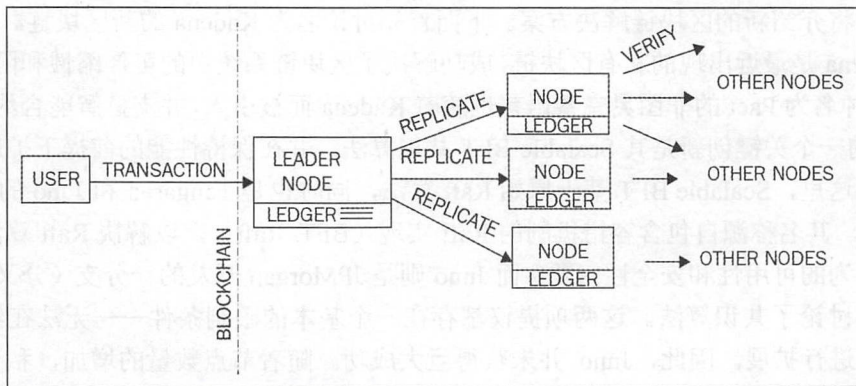
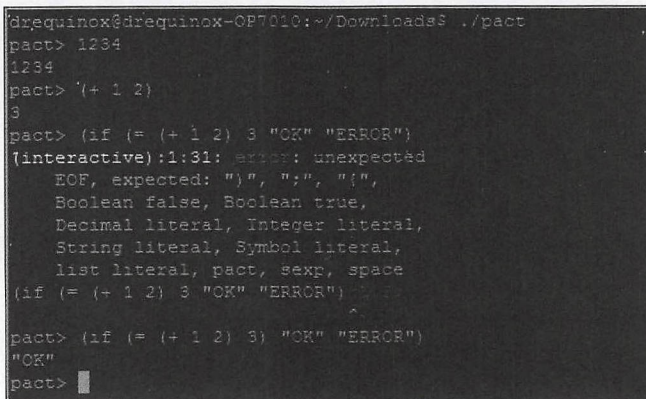


图 10.1 Kadena 中的共识机制

一旦达成共识，即启动智能合约的执行过程，对应步骤如下所示：

- ❑ 首先验证消息签名。
- ❑ Pact 智能合约层负责接管。
- ❑ 编译 Pact 代码。
- ❑ 初始化交易并执行嵌入在智能合约中的业务逻辑。如果出现任何故障，则会立即启动回滚操作，使当前状态恢复至执行开始前的状态。
- ❑ 最后，交易完成且相关日志被更新。

Pact 已经由 Kadena 开源，读者可访问 <http://kadena.io/pact/downloads.html> 进行下载，并将其下载为独立的二进制文件，以提供基于 Pact 语言的 REPL。在图 10.2 中，可输入 `./pact` 命令并在 Linux 控制台中运行 Pact。



```
drequinox@drequinox-QF7010:~/Downloads$ ./pact
pact> 1234
1234
pact> (+ 1 2)
3
pact> (if (= (+ 1 2) 3) "OK" "ERROR")
[interactive]:1:31: error: unexpected
EOF, expected: " ", " ", "{",
Boolean false, Boolean true,
Decimal literal, Integer literal,
String literal, Symbol literal,
list literal, pact, sexp, space
(if (= (+ 1 2) 3) "OK" "ERROR")
~
pact> (if (= (+ 1 2) 3) "OK" "ERROR")
"OK"
pact>
```

图 10.2 Pact REPL，同时显示了示例命令和错误信息

Pact 中的智能合约通常由 3 部分组成：键集、模块和表。首先，键集为表和模块定义了相关的授权方案；其次，模块定义了智能合约代码（包含了函数和 Pact 形式的业务逻辑）。模块内的 Pact 由多个步骤构成并按顺序执行。

Pact 可以在多个执行模式中使用，对应模式包括合约定义、交易执行和查询。其中，合约定义模式允许在区块链上通过单次交易消息创建契约。交易执行模式需要执行智能合约代码模块（表示业务逻辑）；而查询关心的仅是智能合约的数据探查，出于性能考虑，将在节点上以本地方式执行。

协议使用了类 LISP 语法，体现了区块链中所支持的代码内容——对应语法以可阅读的格式存储在区块链中。这与以太坊的 EVM 有所不同，后者编译为可执行的字节代码，因此难以验证在区块链上执行的代码内容。此外，该协议不具备图灵完备性，支持不可变的变量，且不允许使用 null 值，这提高了交易代码执行的总体安全性。

限于篇幅,本章不可能涵盖 Pact 的完整语法和功能。图 10.3 显示了 Pact 合约的一般结构。该示例展示了一个简单的加法模块,定义了一个名为 addition 的函数。addition 函数包含 3 个参数,在执行代码时,将添加 3 个值并显示结果。

```
1- (begin-tx) 'testTransaction ;Begin transaction with optional NAME.  
2 ;Set transaction data in JSON format or pact types  
3 (env-data { "keyset": { "keys": ["admin"] , "pred": "keys-any" } })  
4 ;Define keyset as NAME with KEYSET  
5 (define-keyset 'admin-keyset (read-keyset "keyset"))  
6 ;Set transaction signature KEYS.  
7 (env-keys "admin")  
8 ;define module using syntax (module NAME KEYSET [DOCSTRING] DEFS...)  
9 (module additionModule 'admin-keyset  
10   ;define function that takes three arguments x y z  
11   (defun addition (x y z) (+ x (+ y z)))  
12 )  
13 (commit-tx) ;Commit transaction.  
14 ;use the function addition  
15 (use 'additionModule)  
16 ;run the function addition  
17 (format "Result : {}" (addition 100 200 300))
```

图 10.3 Pact 示例代码

<http://kadena.io/try-pact/>中包含了一个在线编译器示例。当执行代码时,将生成如图 10.4 所示结果。

```
Begin Tx 1  
testTransaction  
Setting transaction data  
Keyset defined  
Setting transaction keys  
Loaded module "additionModule"  
Commit Tx 1  
Using "additionModule"  
Result : 600
```

图 10.4 代码的输出结果

在当前示例中,执行过程的输出结果与代码布局 and 结构完全匹配,从而提高了透明度,并可限制恶意代码的执行。

Kadena 是一种新兴的私有区块链,引入了普遍决定论 (pervasive determinism) 这一新概念。除了标准的公有/私有密钥数据源安全之外,还提供了完全确定性共识这一附加层,进而为区块链的所有层提供加密安全,包括交易和共识层。



注意:

关于 Pact 的相关文档和源代码,读者可访问 <https://github.com/kadena-io/pact> 获得。

1. Ripple

Ripple 是一种货币交换和实时全额结算系统，并于 2012 年推出。在 Ripple 中，支付结算采用实时方式处理，而不是传统的结算网络，后者一般需要几天时间方可完成。另外，Ripple 网络中还发行了一种称为 Ripples (XRP，瑞波币) 的本地货币，同时还支持非 XRP 支付。该系统类似于传统的转账机制 Hawala，其中代理从发送者处获取现金和密码，随后联系收款代理人，并向可以提供密码的人员发放资金。接下来，收款人与本地代理联系，告知其密码并收集资金。这里，代理人类似于 Ripple 中的网关。当然，这只是一个很简单的类比，实际的协议则相当复杂，但基本原理是相同的。

Ripple 网络由不同的节点组成，可以根据其具体类型执行不同的功能。首先是用户节点，此类节点在支付事务中使用，可以支付或接收费用。其次是验证器节点，这些节点参与共识机制。其中，每个服务器维护一组独特的节点，在实现共识机制时需要执行查询操作。唯一节点列表 (UNL) 中的节点是由参与共识机制的服务器所信任的，只接受来自该列表的投票。在涉及网络运营商和监管机构的情况下，某些时候，Ripple 并未视作具有去中心化特征。然而，考虑到任何人都可以通过运行验证器节点成为网络的一部分，因而 Ripple 仍可视作是去中心化的。此外，共识的处理过程也具备去中心化特征——账本中提出的任何改变都必须通过多数表决这一方式来决定。在研究者和爱好者中，这一话题目前十分热门，不同学派之间也持有不同的观点。

Ripple 维护一个包含全部交易的全局分布式账本，并由一种新型的低延迟共识算法控制，即 Ripple 协议共识算法 (RPCA)。该共识处理过程的工作方式可描述为：通过迭代方式搜索验证，并从验证服务器中接收结果（直至获取足够数量的选票），最终实现开放账本的状态协议。一旦获得足够的选票（即大多数选票，最初为 50%，并在每次迭代后逐渐增加，直至 80%），修改内容就会被验证，同时关闭账本。此时，将向全网发送一条警告消息，指示该账本已被关闭。

综上所述，共识协议是一个 3 阶段处理过程，如图 10.5 所示。首先是收集阶段，验证节点收集由账户所有者在网络上广播的所有交易，对其进行验证。交易一旦被接受，即称作候选交易，随后根据验证标准接受或拒绝。接下来启动共识处理过程，操作结束后即关闭账本。该过程每隔几秒就会以异步方式运行，因此，账本会随之处于打开和关闭（更新）状态。

在 Ripple 网络中，存在多个组件可协同工作，以达成共识机制并形成支付网络。下面分别讨论这些组件。

- 服务器。该组件可视为共识协议的参与者。为了能够参与共识协议，需要使用到相关 Ripple 服务器软件。

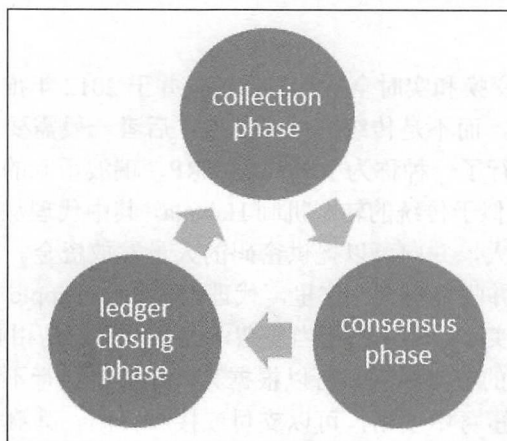


图 10.5 Ripple 共识协议中的各个阶段

- ❑ 账本。表示网络上所有账户余额的主记录。账本包含各种元素，如账本号、账户设置、交易、时间戳和指示账本有效性的标志。
- ❑ 最后关闭的账本。一旦节点验证完毕并达成共识，即关闭账本。
- ❑ 开放式账本。表示尚未经过验证的账本，目前尚未就其状态达成一致。其中，每个节点都包含自己的开放式账本，其中涵盖了将提议的交易内容。
- ❑ 唯一节点列表。表示验证服务器使用的、唯一可信节点的列表，以获取投票和后续共识。
- ❑ 提议者。顾名思义，该组件在共识处理过程中纳入新交易，通常表示为节点子集（之前定义的 UNL），此类节点表示提交至验证服务器的交易。

2. 交易

交易由网络用户所创建，以便更新账本。为了使其在共识处理过程中被选做候选人，交易须经过数字签名且保持有效。相应地，每笔交易都会消费少量的 XRP，作为一种保护机制，这可杜绝垃圾邮件。Ripple 网络中存在不同类型的交易。Ripple 交易数据结构中的单个字段称为 **TransactionType**，用于表示交易的类型。此处，交易是通过 4 个步骤执行的。首先，交易的准备方式可描述为：根据相关标准创建未签名的交易。第二步是签名，期间，交易经数字签名后授权。在此之后，通过连接服务器将其提交至网络。最后，执行验证操作以确保对交易进行成功验证。

粗略地讲，交易可以分为 3 种类型，即支付相关、订单相关、账户和安全相关。

支付相关这一类别中的若干字段可导致执行某些操作，如下所示。

- ❑ **Payment**：该交易较为常用，支持用户间的转账。

- ❑ **PaymentChannelClaim**: 用于从支付通道中请求 Ripple (XRP)。作为一种机制, 支付渠道允许在双方之间进行反复性的单向支付, 同时也可用于设置支付通道的过期时间。
- ❑ **PaymentChannelCreate**: 该交易创建一个新的支付通道, 并将 XRP 添加至其 drop 中。这里, drop 相当于 XRP 的 0.000001。
- ❑ **PaymentChannelFund**: 该交易用于向现有通道注入更多的资金。类似于 PaymentChannelClaim 交易, 也可用来修改支付通道的过期时间。

订单相关交易类型包含以下两个字段。

- ❑ **OfferCreate**: 该交易代表了一种限价单, 代表了货币兑换的意图。如果不能完全实现, 它将导致在协商一致分类账中创建一个提供节点。
 - ❑ **OfferCancel**: 用于从共识账本中删除之前创建的节点, 以表明该订单退订。
- 账户和安全相关类型的交易包括如下所列的字段, 每个字段负责执行某个函数。
- ❑ **AccountSet**: 用于修改 Ripple 共识账本中账户的属性。
 - ❑ **SetRegularKey**: 用于更改或设置账户的交易签名密钥。使用从账户主公钥派生的 base-58 Ripple 地址来标识账户。
 - ❑ **SignerListSet**: 针对多签名交易应用, 可创建一组签名者。
 - ❑ **TrustSet**: 用于创建或修改账户之间的信任机制。

Ripple 中的交易由多个字段组成, 并适用于所有交易类型, 如下所示:

- ❑ **Account** 表示交易发起者的地址。
- ❑ 作为可选的字段, **AccountTxnID** 包含了另一项交易的哈希值。
- ❑ **Fee** 表示 XRP 的数量。
- ❑ **Flags** 表示交易的可选标记。
- ❑ **LastLedgerSequence** 表示交易所显示的账本中的最高序列号。
- ❑ **Memos** 表示可选的任意信息。
- ❑ **Sequence** 表示每笔交易所增加的数字 (通常为 1)。
- ❑ **SigningPubKey** 表示公钥。
- ❑ **Signers** 表示多重签名交易中的签名者。
- ❑ **SourceTag** 表示交易的发送者或相关原因。
- ❑ **TransactionType** 表示交易的类型。
- ❑ **TxnSignature** 表示交易的验证签名。

为了使外部实体能够连接到 Ripple 网络, 开发人员提供了各种相关的 API。作为关键组件, Interledger 协议和 Ripple 连接支持分布式、安全、可扩展和互操作的支付网络。

Interledger 协议支持两种不同账本之间的互操作性,可用于连接各种不同组织机构的账本和区块链,包括但不限于支付网络、金融机构、票据交换机构和交易机构。

Interledger 是一种简单的协议,由 4 层组成,即 Application 层、Transport 层、Interledger 层以及 Ledger 层。每层负责在特定协议下执行各种功能。

- ❑ Application 层。该层上运行的协议负责管控支付交易的关键属性,对应示例包括简单的支付设置协议 (SPSP) 和开放的 Web 支付方案 (OWPS)。作为一种 Interledger 协议,SPSP 可在不同的账本之间创建连接器以实现安全支付。OWPS 则是另一种允许用户在不同网络上支付的方案。一旦该层协议成功运行,将调用源自传输层中的协议以启动支付流程。
- ❑ Transport 层。该层负责管理支付交易,Optimistic Transport 协议 (OTP)、Universal Transport 协议 (UTP) 和 Atomic Transport 协议 (ATP) 等目前都适用于这一层。OTP 是最简单的协议,负责管理没有任何托管保护的支付传输,而 UTP 则提供了托管保护。ATP 则是一种较为先进的协议,不仅提供了托管传输机制,而且还通过受信公证人进一步保障支付交易。
- ❑ Interledger 层。该层提供互操作性和路由服务,其中涉及 Interledger 协议 (ILP)、Interledger 引用协议 (ILQP) 和 Interledger 控制协议 (ILCP) 等。ILP 包在传输过程中提供交易的最终目标。在实际传输之前,ILQP 用于生成发送方的报价请求。在支付网络上的连接器之间,ILCP 用于交换与路由信息和支付错误相关的数据。
- ❑ Ledger 层。该层包含能够在连接器之间通信和执行支付交易的协议。此处,连接器定义为一个对象,并实现了不同账本之间转发支付的协议。Ledger 层支持各种协议,如 Simple Ledger 协议、各种区块链协议、传统协议和不同的专有协议。

Ripple 连接由各种可插接的模块组成,通过 ILP 实现账本之间的连接性,并可在交易、可见性、费用管理、交付确认和安全通信 (利用 Transport 层的安全性) 之间交换所需的数据。相应地,第三方应用程序可以通过各种连接器连接到 Ripple 网络。

总而言之,Ripple 是一项针对金融行业的解决方案,并降低实时支付中的任何结算风险。该平台涵盖了丰富的内容,因而本章不可能面面俱到。关于 Ripple 及其文档,读者可访问 <https://ripple.com/> 以获取更多信息。

3. Stellar

Stella 是基于区块链技术的支付网络,同时也是一种称为联邦拜占庭协议 (FAB) 的新型共识模型。FBA 通过创建受信任各方的法定人数来工作;而 Stella 共识协议 SCP 则是 FBA 的一个实现。

Stella 白皮书中指出关键问题是当前金融基础设施的成本和复杂度问题。在不损害金融交易的完整性和安全性的前提下，当全球金融网络尝试解决此类问题时，这种限制充分体现了相关必要性。在此基础上，产生了 Stella 共识协议（SCP），经证明，这是一种相对安全的共识机制。

Stella 包含 4 种主要属性：去中心化控制（允许任何人参与，且不存在中心机构）；低延迟（满足了快速交易处理这一需求）；灵活的信任机制（针对特定目标，允许用户选择受信方）；以及渐近式安全性（利用数字签名和哈希函数提供网络所需的安全级别）。

Stella 网络允许通过其原生数字货币 Lumens 传输和表示资产的价值，简称 XLM。当交易在网络上进行广播时，就会消耗大量的 Lumens，这也可以作为对拒绝服务（DOS）攻击的一种威慑。

作为核心内容，Stella 网络维护一个分布式账本，记录每一项交易，并在每个 Stella 服务器上复制；同时，可通过验证服务器之间的交易和更新账本达成共识；另外，用户可存储其认购或货币抛售内容，据此，Stella 账本还可以表示为一个分布式交易订单簿。

Stella 网络涉及各种工具、SDK 以及软件。读者可访问 <https://github.com/stellar/stellar-core> 获取其核心软件。

4. Rootstock

在详细讨论 Rootstock 之前，需要定义并引入某些概念，这些概念对于 Rootstock 的设计是至关重要的。相关概念包括侧链、驱动链和双向锚定。其中，侧链这一概念最初是由 Blockstream 提出的。

通过双向锚定机制，价值（货币）可在区块链之间转移，反之亦然。需要注意的是，链之间并不存在真正的货币转移。这一理念主要围绕着以下概念展开：锁定比特币区块链（主链）中等量、等额的货币，并解锁二级链中同等数量的代币。

侧链是一个与主区块链并行运行的区块链，二者间可实现传递值。这意味着，区块链中的代币可以在侧链中使用，反之亦然。由于支持双向锚定资产，因而也被称作锚定链。

5. 驱动链

驱动链则是一个相对较新的概念，将锁定比特币（在主链中）的解锁控制权交给能够投票的矿工，并在解锁时进行投票。这与侧链形成了鲜明的对比，此时，共识通过简单的支付验证机制进行验证，以便将货币转移回主链。

Rootstock 是一个智能合约平台，并与比特币区块链具有双向锚定，其核心理念是提高比特币系统的可扩展性和性能，使其能够与智能合约协同工作。另外，Rootstock 运行一个名为 Rootstock 虚拟机（RVM）的图灵完全确定性虚拟机。Rootstock 还与以太坊虚

虚拟机兼容，可在 Rootstock 上运行 Solidity 编译的契约。智能合约还可以在比特币区块链的 time-a 安全测试环境下运行。Rootstock 区块链的工作原理是将挖掘和比特币合并，允许 RSK 区块链实现与比特币相同的安全级别，这对于防止双重支出和实现最终结算尤其正确。此外，Rootstock 还具备可扩展性，且每秒多达 100 项事务。

RSK 最新发布了称为 Turmeric 的测试网络，读者可访问 <http://www.rsk.co/> 获取更多信息。

6. Quorum

作为一种区块链解决方案，Quorum 通过增强现有的 Ethereum 区块链来构建，其增强性体现在交易隐私和新的共识机制。Quorum 引入了一种新的共识模型 QuorumChain，且基于多数投票和时间机制。另外，Quorum 还包含了一种 Constellation 功能，它是一种提交信息的通用机制，允许用户之间进行加密通信。此外，节点级别上的授权许可机制由智能合约管理。与公共以太坊区块链相比，Quorum 还提供了更高的性能级别。

区块链生态系统包含了多个组件，如下所示：

- ❑ 交易管理器。该组件允许访问加密的交易数据，同时管理与网络上其他交易管理器的本地存储和通信。
- ❑ 加密 Enclave。顾名思义，该组件负责提供加密服务以确保交易隐私，以及执行密钥的管理功能。
- ❑ QuorumChain。这也是 Quorum 中的一项关键创新内容。作为拜占庭容错共识机制，允许通过区块链网络上的交易实现验证和选票流通。在该方案中，智能合约用于管理共识过程，节点可以获得投票权，以投票决定哪个新区块应该被接受。一旦投票者获得一定的投票数量，对应区块即认为有效。这里，节点可以饰演两个角色，即 Voter 或 Maker。其中，Voter 节点具有投票权利，而 Maker 节点则创建一个新的区块。
- ❑ 网络管理器。针对授权网络，该组件提供了访问控制层。

Quorum 网络中的一个节点可以扮演多个角色，例如创建新区块的 Maker 节点。此处，交易隐私是使用加密技术提供的，而某些交易只能由相关参与者来查看。这一理念与第 9 章讨论的 Corda 私人交易不谋而合。由于区块链支持公有和私有事务，因而状态数据库被划分为两种数据库，分别表示私有和公有交易。因此，存在两棵独立的 Patricia-Merkle 树，代表了网络的私有和公有状态。另外，私有合约状态哈希值用于在交易双方之间的私有交易中提供一致的证据。

在 Quorum 网络中，交易由各种元素构成，例如接收者、发送者的数字签名（用于确定交易的发起者）、可选的 Ether 数量、可选的参与者列表（可以查看当前交易），以及

包含私有交易哈希值的某个字段。

交易须经过多个步骤方可到达最终目的地，如下所示：

- ❑ 用户应用程序（DAPP）通过区块链网络公开的 API 将交易发送至仲裁节点，其中也包含接收者地址和交易数据。
- ❑ 随后，API 对有效负载内容进行加密，并执行所需的加密算法，以确保交易的隐私性，最后被发送至交易管理器。除此之外，加密负载的哈希值也在该步骤中计算。
- ❑ 在交易被接收后，交易管理器将对交易发送者的签名进行验证，并存储对应消息。
- ❑ 之前加密有效负载内容的哈希值将发送至 Quorum 节点中。
- ❑ 一旦 Quorum 节点开始验证包含私有交易的区块，即会从交易管理器中请求更多的相关数据。
- ❑ 当交易管理器接收此请求后，将加密的有效负载和相关的对称密钥发送到请求者仲裁节点。
- ❑ 当 Quorum 节点具备所有数据后，即解密有效负载内容并将其发送至 EVM 执行。这体现了 Quorum 在区块链上基于对称加密的隐私实现方式，同时还可以分别使用本地以太坊协议和 EVM 进行消息传递和执行操作。
- ❑ 类似概念曾以 Hydrachain 这一形式提出，但在某些方面仍存在较大的不同。Hydrachain 在以太坊区块链的基础上构建，并可生成授权许可的分布式账本。

读者可访问 <https://github.com/jpmorganchase/quorum> 下载 Quorum。

7. Tezos

Tezos 是一种通用的自我修正型加密账本，这意味着，可在区块链的状态上支持去中心化的共识，而且还允许在协议和节点如何随时间的变化这一问题上达成共识。Tezos 旨在解除比特币协议中的种种限制，例如硬分叉、成本问题、工作量证明导致的挖掘中心化问题、有限的脚本功能，以及安全问题。Tezos 可通过 OCaml 这一纯函数语言进行开发。

Tezos 分布式账本架构进而分为 3 层，即网络层、共识层和交易层。这种分解方式允许协议以一种去中心化的方式演变。为此，Tezos 中实现了一个通用的网络 shell（网壳），负责维护区块链，并由共识和事务层这一组合方式加以表示。此处，shell 提供了网络和协议之间的接口层。除此之外，Tezos 还引入了种子（seed）协议这一概念。作为一种机制，该协议允许利益相关者在网络上对协议进行更改。Tezos 区块链源自种子协议，这一点与传统的区块链不同，后者源自创始区块。

种子协议负责定义区块链中的修正程序，甚至可修改协议本身。另外，Tezos 中的奖励机制基于权益证明（PoS）算法，因此不存在任何挖掘需求。

Tezos 中也引入了合约脚本语言，并可用于编写智能合约，该语言是一种基于堆栈的图灵完备语言。Tezos 中的智能合约具有可验证特征，也就是说，代码在数学上被证明是正确的。

读者可访问 <https://github.com/tezos/tezos> 下载 Tezos。

8. Storj

基于云存储的现有模型均为集中式解决方案，可能缺乏应有的安全性。对此，需要配备一种安全的、高度可用的云存储系统，尤其是去中心化的云存储系统。因此，Storj 的目标是提供一类基于区块链的、去中心化的分布式存储。Storj 是一种社区共享云，而不是某个中心组织；并可执行自治代理节点之间的存储合约。此类代理（节点）可执行各种功能，例如数据传输、验证和数据完整性检查。Storj 核心概念基于 DHT（分布式哈希表）-Kademlia 协议，该协议通过在 Storj 中添加新的消息类型和功能得到了增强。另外，Storj 还实现了 P2P 发布/订阅（pub/sub）机制，称为 Quasar，可确保消息成功地到达对存储合约感兴趣的节点。该过程是通过 Bloom 过滤器存储合约参数选择机制实现的，即 topics。

采用加密格式的 Storj 存储文件在网络上传播，在该文件存储在网络中之前，采用了 AES-256-CTR 对称加密方式进行加密，然后以分布方式逐一存储在网络上。这里，文件的分解过程称为分片，并提高了网络的可用性、安全性、性能和隐私性。另外，如果某一节点失效，分片仍然可用——默认情况下，某个分片存储在网络上的 3 个不同位置。

Storj 维护一个区块链，可视为一个共享账本，并实现了标准的安全特性，如公有/私有密钥加密，以及与其他区块链类似的哈希函数。由于该系统采用了对等点的硬盘共享机制，因而任何人都可以分享其硬盘上的额外空间，并获得 Storj 自己的加密货币 Storjcoinx (SJCX)。SJCX 是作为交易对手方的资产而开发的，并利用比特币区块链进行交易。

读者可访问 <https://github.com/Storj>/获取 Storj 代码。

9. Maidsafe

Maidsafe 是另一个与 Storj 类似的分布式存储系统。用户通过对网络的存储空间贡献获取 Safecoin。这种支付机制采用资源证明加以管理，确保了用户对网络所提交的磁盘空间是可用的；否则，Safecoin 支付也将随之减少。这些文件经过加密后被划分为多个组成部分，然后再传输至网络中予以存储。除此之外，Maidsafe 还引入了另一种时机性缓存概念，可以创建频繁访问的数据的副本，从而更接近访问请求的来源，进而提高网络的性能。

SAFE 网络的另一个新特性是，可自动删除网络上的任何重复数据，从而降低存储需求。此外，该网络还引入了“搅动”这一概念，也就是说，数据在网络上不断地移动，以便数据无法被恶意攻击者所定位；同时，还可以在整个网络中保存多个数据副本，以便在节点脱机或无效时提供冗余内容。

10. BigChainDB

BigChainDB 是一个可扩展的区块链数据库，本身并不是一个区块链，而是通过提供一个去中心化的数据库以对区块链技术进行有益的补充。BigChainDB 的核心是一个分布式数据库，但增加了区块链的属性，如去中心化、不变性和数字资产处理。同时，BigChainDB 还支持 NoSQL 来查询数据库，其目的是在去中心化系统中提供一个数据库，在该生态系统中，处理过程（区块链）、文件系统（例如 IPFS）以及数据库均处于去中心化状态，因而最终应用程序生态系统也处于去中心化状态。读者可访问 <https://www.bigchaindb.com/> 下载 BigChainDB。

11. 多重链

对于私有区块链的开发和部署，多重链已形成了一个平台，该平台基于比特币代码，并解决安全性、可扩展性和隐私问题。作为可配置的区块链平台，多重链允许用户设置不同的区块链参数，通过许可层来支持控制和隐私操作。多重链的安装过程较为简单，读者可访问 <http://www.multichain.com/download-install/> 获取安装链接文件。

12. Tendermint

Tendermint 是一个软件，为应用程序提供了拜占庭式的容错共识机制和状态机复制功能，旨在开发通用、安全、高性能的复制状态机。

Tendermint 包含两个组件，如下所示：

- ❑ Tendermint Core。作为共识引擎，Tendermint Core 可以在网络的每个节点上启用安全的交易复制过程。
- ❑ Tendermint 套接字协议（TMSP）。TMSP 是一个应用程序接口协议，允许与任何编程语言进行交互以对交易进行处理。

Tendermint 支持应用程序处理和共识处理的解耦操作，这使得任何应用程序都可以从共识机制中获益。

Tendermint 共识算法采用回合（round）机制，验证器节点在每个回合中产生新区块。另外，还使用锁定机制针对以下情形进行保护：在区块链相同高度上选择两个不同区块并提交。每个验证器节点维护一个完整的本地复制区块（包含当前交易）；其每个区块包含一个头、之前区块哈希值、区块形成时的时间戳、当前区块高度，以及区块中所有交易的 Merkle 根哈希值。

Tendermint 最近被应用于 Cosmos 中，作为区块链的网络，Cosmos 支持运行于 BFT 共识算法上不同链之间的互操作性，该网络上的区块链称作区域（zone）。Cosmos 中的第一个区域称为 Cosmos Hub，实际上是一个公共区块链，负责为其他区块链提供连接服

务。为此，Cosmos Hub 使用了内部区块链通信协议（IBC）。IBC 协议支持 IBCBlockCommitTx 和 IBCPacketTx 两种类型的交易。第一类用于向任何一方提供区块链中最近的区块哈希证明，而后者则用于提供数据源身份验证。区块链间数据包的发送方式可描述为：首先，可向目标链发布一个证明，随后接收（目标）链检查该证明，以验证发送链确实已发布该数据包。另外，该区块链包含自己的本地货币 Atom。通过多个区块链连接至 Hub，该方案解决了扩展性和互操作性问题。

读者可访问 <https://tendermint.com/> 下载 Tendermint。

10.2 平 台

本节涵盖了当前各种开发平台，以对现有区块链解决方案予以必要的补充。下面首先讨论名为 BlockApps STRATO 的以太坊兼容解决方案。

10.2.1 BlockApps

BlockApps 平台提供了丰富的工具构建区块链应用程序，并采用 Haskell 语言编写且基于模块化架构。该解决方案具有可扩展性，可以更容易地部署智能合约和区块链应用程序。读者可访问 <http://www.blockapps.net/> 下载 BlockApps。

下面考察 BlockApps 的安装过程及其简单的部署示例。

1. 安装

BlockApps 可通过 npm 进行安装，对应命令如下所示：

```
$ sudo npm install -g blockapps-bloc
```

其中，sudo 为可选项（如果不需要管理权限）。输出结果如图 10.6 所示。

```
npm WARN deprecated secp256k1-browserify@0.0.0: secp256k1 now includes browser components
/usr/bin/bloc -> /usr/lib/node_modules/blockapps-bloc/bin/main.js
/usr/lib
├─┬─ blockapps-bloc@1.2.2
│   ├── bignumber.js@2.4.0
│   └─┬─ blockapps-js@3.1.2
│       ├── bn.js@4.11.6
│       ├── elliptic@6.3.2
│       ├── brorand@1.0.6
│       ├── hash.js@1.0.3
│       └─┬─ inherits@2.0.3
```

图 10.6 通过 npm 安装 BlockApps（输出结果有所省略）

安装完成后，可以按照后续内容所示步骤创建应用程序。下面的例子展示了如何在 BlockApps 中初始化一个新的应用程序、BlockApps TestNet 中的部署和交互方式。

2. 应用程序部署和 BlockApps 部署

首先需要通过下列命令初始化 BlockApps:

```
$ bloc init
```

这将请求多个参数，包括应用程序名称、用户名称、电子邮件 API URL (apiUrl) 以及区块链配置文件。对应结果如图 10.7 所示。

```
drequinox@drequinox-0P7010:~$ bloc init
? ~~~~~
We're constantly looking for ways to make blockapps-bloc better!
May we anonymously report usage statistics to improve the tool over time?
More info: https://github.com/blockapps/bloc & http://blockapps.net
~~~~~ No

BlockApps

prompt: Enter the name of your app: testApp
prompt: Enter your name: drequinox
prompt: Enter your email so BlockApps can reach you:
prompt: apiURL: (http://strato-dev4.blockapps.net)
prompt: Enter the blockchain profile you wish to use. Options: strato-dev, ethereum: (strato-dev)
Wrote: /home/drequinox/testApp/.bowerrc
Wrote: /home/drequinox/testApp/app.js
Wrote: /home/drequinox/testApp/bower.json
Wrote: /home/drequinox/testApp/gulpfile.js
Wrote: /home/drequinox/testApp/marko-taglib.json
Wrote: /home/drequinox/testApp/package.json
Wrote: /home/drequinox/testApp/test/common.js
Wrote: /home/drequinox/testApp/test/top.js
Wrote: /home/drequinox/testApp/app/contracts/Greeter.sol
Wrote: /home/drequinox/testApp/app/contracts/MultiContract.sol
Wrote: /home/drequinox/testApp/app/contracts/Payout.sol
```

图 10.7 初始化过程

当命令运行并成功结束后，将生成包含模板和示例的应用程序目录。在当前示例中，将生成名为 testApp 的目录，其中包含了相关目录和示例合约。

testApp 的安装可采用下列命令：

```
$ sudo npm install
```

对应结果如图 10.8 所示。

签署交易时需要生成一个新的密钥，并可使用以下命令完成：

```
$ bloc genkey
```

此处需要输入密码以保护密钥，随后将创建密钥以及一个 JSON 文件。注意，JSON 文件名是区块链上账户的实际地址。此外，还将显示与（被挖掘的）交易相关的消息，

表示密钥和交易（账户创建）的成功操作和部署结果。

```
drequinox@drequinox-OP7010: ~/testApp
drequinox@drequinox-OP7010:~/testApp$ sudo npm install
[sudo] password for drequinox:
npm WARN deprecated secp256k1-browserify@0.0.0: secp256k1 now includes browser components
npm WARN deprecated minimatch@2.0.10: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
npm WARN deprecated minimatch@2.0.14: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
npm WARN deprecated graceful-fs@1.2.3: graceful-fs v3.0.0 and before will fail on node releases >= v7.0. Please update to graceful-fs@4.0.0 as soon as possible. Use 'npm ls graceful-fs' to find it in the tree.
npm WARN deprecated to-iso-string@0.0.2: to-iso-string has been deprecated, use @segment/to-iso-string instead.
npm WARN deprecated jade@0.26.3: Jade has been renamed to pug, please install the latest version of pug instead of jade
npm WARN deprecated minimatch@0.3.0: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue

> gulp-express@0.3.5 install /home/drequinox/testApp/node_modules/gulp-express
> echo "*** Please use [gulp-live-server] instead! ***"

*** Please use [gulp-live-server] instead! ***
npm WARN lifecycle testApp@1.0.0-postinstall: cannot run in wd %s %s (wd=%s) testApp@1.0.0 node node_modules/bower/bin/bower install /home/drequinox/testApp
testApp@1.0.0 /home/drequinox/testApp
├─┬ blockapps-js@3.1.2
├─┬ bignumber.js@2.4.0
└─┬ bluebird@2.11.0
```

图 10.8 安装 testApp

上述处理过程如图 10.9 所示。

```
drequinox@drequinox-OP7010: ~/testApp
drequinox@drequinox-OP7010:~/testApp$ bloc genkey
prompt: Enter a high entropy password. You will need this to sign transactions.:
wrote app/users/admin/b16fe63de8fcf9d97f1d787cba37a02309104316.json
transaction successfully mined!
drequinox@drequinox-OP7010:~/testApp$
```

图 10.9 创建密钥

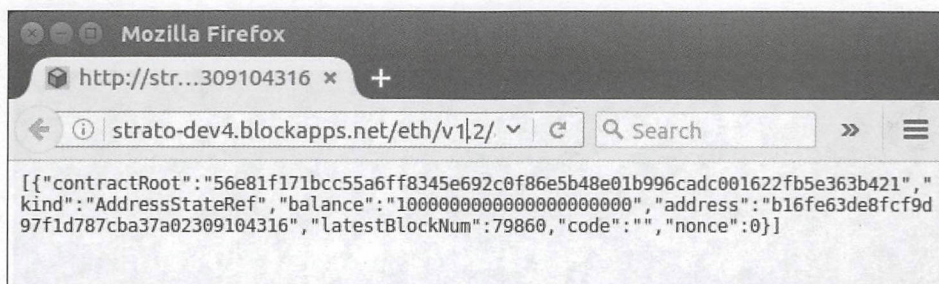
当前，可以使用 curl 来查询新账户。对此，只需将地址作为 URL 中的参数传递，结果将以 JSON 格式返回，如图 10.10 所示。

```
drequinox@drequinox-OP7010:~/testApp$ curl http://strato-dev4.blockapps.net/eth/v1.2/account?address=b16fe63de8fcf9d97f1d787cba37a02309104316
[{"contractRoot":"56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421","kind":"AddressStateRef","balance":"10000000000000000000000","address":"b16fe63de8fcf9d97f1d787cba37a02309104316","latestBlockNum":79860,"code":"","nonce":0}]drequinox@drequinox-OP7010:~/testApp$
```

图 10.10 通过 curl 查询新账户

除此之外，查询操作还可通过 Web 浏览器完成，如图 10.11 所示。

下一步将考察如何将新合约上传至测试链。需要注意的是，所有的合约均置于 testApp 目录下的 ./app/contracts 目录中。作为示例，此处选择了 Greeter.sol 部署至网络上。对此，BlockApps 提供了一个简单的方法来实现这一部署过程。



10.11 通过 Web 浏览器查询 BlockApps

全部合约需要置于 contracts 目录下，以供编译命令搜索和编译，如图 10.12 所示。

```

drequinox@drequinox-OP7010: ~/testApp/app/contracts
drequinox@drequinox-OP7010:~/testApp/app/contracts$ pwd
/home/drequinox/testApp/app/contracts
drequinox@drequinox-OP7010:~/testApp/app/contracts$ ll
total 40
drwxrwxr-x 2 drequinox drequinox 4096 Dec 27 10:24 ./
drwxrwxr-x 8 drequinox drequinox 4096 Dec 27 10:45 ../
-rw-rw-r-- 1 drequinox drequinox 695 Dec 27 10:24 Greeter.sol
-rw-rw-r-- 1 drequinox drequinox 237 Dec 27 10:24 MultiContract.sol
-rw-rw-r-- 1 drequinox drequinox 618 Dec 27 10:24 Payout.sol
-rw-rw-r-- 1 drequinox drequinox 178 Dec 27 10:24 SimpleDataFeed.sol
-rw-rw-r-- 1 drequinox drequinox 1421 Dec 27 10:24 SimpleMultiSig.sol
-rw-rw-r-- 1 drequinox drequinox 181 Dec 27 10:24 SimpleStorage.sol
-rw-rw-r-- 1 drequinox drequinox 998 Dec 27 10:24 Stake.sol
-rw-rw-r-- 1 drequinox drequinox 663 Dec 27 10:24 template.marko
drequinox@drequinox-OP7010:~/testApp/app/contracts$ cat Greeter.sol
contract mortal {
    /* Define variable owner of the type address*/
    address owner;

    /* this function is executed at initialization and sets the owner of the contract */
    function mortal() { owner = msg.sender; }

    /* Function to recover the funds on the contract */
    function kill() { if (msg.sender == owner) suicide(owner); }
}

contract Greeter is mortal {
    /* define variable greeting of the type string */
    string greeting;

    /* this runs when the contract is executed */
    function Greeter(string _greeting) public {
        greeting = _greeting;
    }

    /* main function */
    function greet() constant returns (string) {
        return greeting;
    }
}
drequinox@drequinox-OP7010:~/testApp/app/contracts$

```

图 10.12 位于 contracts 目录下的 Greeter 合约

图 10.13 显示了编译合约时使用的命令。注意，该命令将合约文件名作为参数。成功

编译后，将在./meta 目录下编写所有相关的 JSON 文件。

```
drequinox@drequinox-0P7010:~/testApp$ bloc compile Greeter
Compiling single contract: Greeter.sol
Compile successful: contract mortal {
  /* Define variable owner of the type address*/
  address owner;

  /* this function is executed at initialization and sets the owner of the contract */
  function mortal() { owner = msg.sender; }

  /*Function to recover the funds on the contract */
  function kill() { if (msg.sender == owner) suicide(owner); }
}

contract Greeter is mortal {
  /* define variable greeting of the type string*/
  string greeting;

  /* this runs when the contract is executed */
  function Greeter(string _greeting) public {
    greeting = _greeting;
  }

  /* main function */
  function greet() constant returns (string) {
    return greeting;
  }
}

writing Greeter to app/meta/Greeter/Greeter.json
wrote: app/meta/Greeter/Greeter.json
writing mortal to app/meta/Greeter/mortal.json
wrote: app/meta/Greeter/mortal.json
writing Greeter to app/meta/mortal/Greeter.json
wrote: app/meta/mortal/Greeter.json
writing mortal to app/meta/mortal/mortal.json
wrote: app/meta/mortal/mortal.json
drequinox@drequinox-0P7010:~/testApp$
```

图 10.13 Greeter 合约的编译过程

最后，可使用下列命令上传合同。该命令希望将参数传递给合约代码中定义的合约，当前示例中表示为一个文本字符串，如图 10.14 所示。

```
$ bloc upload Greeter "Hello bloc"
```

```
drequinox@drequinox-0P7010:~/testApp$ bloc upload Greeter "Hello bloc"
address: b16fe63de8fc9d97f1d787cba37a02309104316
prompt: Enter password to retrieve private key:
upload contract: Greeter
writing: app/meta/Greeter/05ee3af04e903f413402d5438b9de3827b1f4e70.json
writing: app/meta/Greeter/Latest.json
creating metadata for Greeter
drequinox@drequinox-0P7010:~/testApp$
```

图 10.14 Greeter 合约的上传过程

注意，如果未传递正确的参数或参数遗失，将会出现如图 10.15 所示的错误消息。

一旦部署成功，即可以验证现有合同转移至新合同之后的 Ether。如图 10.16 所示，当前余额减少。


```
drequinox@drequinox-0P7010:~/test$ npm run upload Greeter
address: b10fe63de8fc9d97f1d787cba37a02309104310
prompt: Enter password to retrieve private key:
upload contract: Greeter
there was an error: {"errorTags":["Solidity","Solidity"],"message":"function \"Greeter\" arguments must include \"_greeting\""}
creating metadata for: Greeter
```

图 10.15 错误参数或参数遗失产生的错误消息

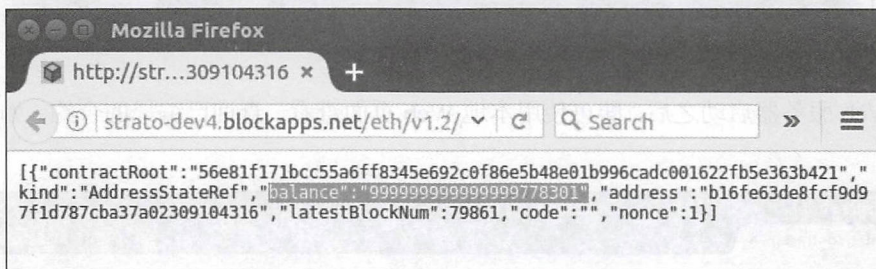


图 10.16 通过 Web 浏览器，安装后所部署的合约

合约部署完毕后，需要通过 Web 浏览器或 CLI 工具对其进行查询，例如 cURL。同时，还需向 Web 浏览器传递 `http://strato-dev4.blockapps.net/eth/v1.2/account?address=05ee3af04e903f413402d5438b9de3827b1f4e70` 这一 URL，如图 10.17 所示。需要注意的是，输出结果中包含了二进制格式的代码。

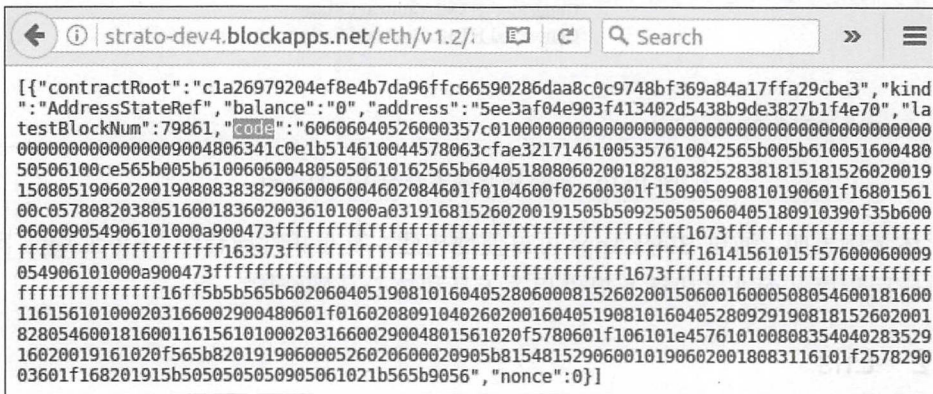


图 10.17 浏览部署后的合约，其中包含了二进制格式的代码

进一步讲, BlockApps 涵盖了一项功能并可运行本地 HTTP 服务器。相应地, 可通过下列命令启用该项功能:

```
$ bloc start
```

这将启动 Web 浏览器, 并在 TCP 端口 8000 上进行监听, 如图 10.18 所示。

```
drequinox@drequinox-OP7010: ~/testApp
drequinox@drequinox-OP7010:~/testApp$ bloc start
bloc is listening on http://0.0.0.0:8000

api is pointed to http://strato-dev4.blockapps.net with profile strato-dev
```

图 10.18 启动 bloc

在 Web 服务器启动之后, 即可使用本地 Web 页面查看、查询已编译的合约, 如图 10.19 所示。

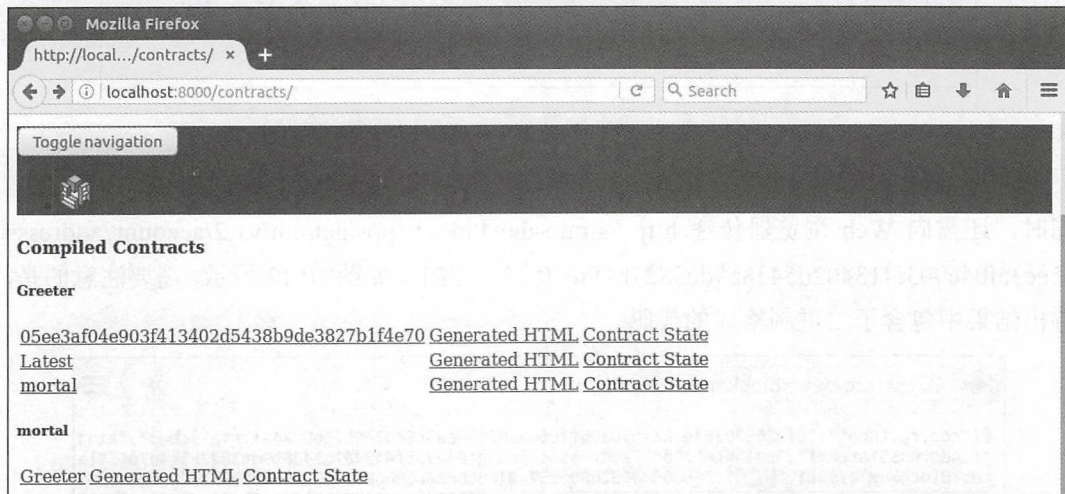


图 10.19 通过浏览器查看编译后的合约

上述示例表明, 通过 BlockApps 可方便地构建、发布和管理合约。BlockApps 旨在向区块链应用程序 (而非区块链) 提供相关工具和核心基础结构。

10.2.2 Eris

Eris 是 Monax 开发的开放式模块化平台, 而非区块链, 用于开发基于区块链的生态

系统应用程序，并提供了各种框架、SDK 和工具，支持快速开发和部署区块链应用程序。Eris 应用程序平台背后的核心思想是使用区块链后台支持生态系统应用程序的开发和管理，同时支持集成多个区块链，并使第三方系统与其他系统进行交互。该平台使用 Solidity 语言编写的智能合约，可以与像以太坊或比特币这样的区块链进行交互。这里，交互过程包括连接命令、启动、停止、断开和创建新的区块链。与区块链的设置和交互相关的复杂操作在 Eris 中均已被抽象出来；对于不同的区块链，所有的命令都是标准化的，同样的命令可实现跨平台使用，且无须考虑区块链的类型。

生态系统应用程序可组建 Eris 平台，同时启用 API 网关，使得遗留应用程序连接到管理系统、共识引擎和应用程序引擎。此外，Eris 平台提供了各种工具包，用于为开发人员提供各种服务。这些模块描述如下。

- ❑ 链：可创建区块链并与其进行交互。
- ❑ 包：用于开发智能合约。
- ❑ 密钥：用于密钥管理和签名操作。
- ❑ 文件：可与分布式数据管理系统协同工作，以及与文件系统进行交互，例如 IPFS 和数据湖泊（data lakes）。
- ❑ 服务：显示了一组服务，以支持生态系统应用程序的管理和集成。

Eris 发布了多种 SDK，进而开发和管理生态系统应用程序。这一类 SDK 包含了经过充分测试并满足业务需求的智能合约，如财务 SDK、保险 SDK 和物流 SDK。除此之外，作为一个基本的开发工具包，基本 SDK 则用于管理生态系统应用程序的生命周期。

Monax 推出了自己的区块链客户端，称为 Eris:db，可视为一种权益证明区块链系统，同时支持与不同的区块链网络进行集成。Eris:db 由 4 个部分组成，如下所示。

- ❑ 共识：基于之前讨论的 Tendermint 共识机制。
- ❑ 虚拟机：Eris 使用以太坊虚拟机（EVM），因而支持 Solidity 编译合约。
- ❑ 授权许可层：作为一种授权许可账本，Eris 提供了一种访问控制机制，进而可向网络上不同的实体赋予不同的角色。
- ❑ 接口：提供了各种命令行工具和 RPC 接口，从而可与后台区块链网络进行交互。

以太坊区块链与 Eris:db 之间的主要差别在于：Eris:db 采用了拜占庭容错算法，且实现为基于存储的权益证明（DPOS 系统）；而以太坊使用了工作量证明（PoW）。除此之外，Eris:db 采用了 ECDSA ed25519 曲线方案，以太坊则使用了 secp256k1 算法。最后，Eris:db 通过访问控制层实现授权许可，而以太坊则定义为公共区块链。

Eris 是一类功能丰富的应用程序平台，提供了多种工具包和服务，并在此基础上开



发基于区块链的应用程序。读者可访问 <https://monax.io/> 下载 Eris。

10.3 本章小结

本章首先介绍了可替代的区块链，并从两个主要方面探讨区块链和平台。目前，区块链技术是一个非常活跃的领域，因而既有方案的变更也较为迅速，而且几乎每天都会涌现出新的技术和相关工具。本章讨论了平台和区块链选取方案，例如支持以太坊开发的 BlockApps、新的区块链如 Kadena，各种新的协议如 Ripple，以及诸如侧链和驱动链等概念。如前所述，区块链是发展迅猛的领域，限于篇幅，本章内容不可能面面俱到，例如 Tauchain、Hydrachain、Elements、credits 等项目。同时，我们也鼓励读者持续关注这一领域的发展，并能够做到与时俱进。



第 11 章 货币之外的区块链技术

数字货币是区块链技术的第一个应用，之前人们并没有意识到其真正的潜力。随着比特币的出现，区块链这一概念被首次引入；直到 2013 年，区块链 2.0 的出现才使区块链的优势在各个行业中得到了应用。从那时起，在不同行业中，区块链技术用例被人们反复提及，包括但不限于金融、物联网、数字版权管理、政府和法律等行业。本章针对物联网（IoT）、政府、健康和金融 4 个主要行业进行讨论。届时，将对这些领域逐一加以考察，并介绍各种相关的用例。

11.1 物 联 网

近期，物联网或 IoT 因其改变商业应用和日常生活行为的能力而获得了广泛的关注。物联网定义为具有计算能力的智能物理对象网络，IoT 能够连接至互联网，感知真实世界的事件或环境，随后对此类事件做出反应，收集相关数据，并通过互联网进行通信。这一简单的定义产生了巨大的影响，并产生了一些令人兴奋的概念，如可穿戴设备、智能家居、智能电网、智能汽车和智慧城市，均可视作基于物联网设备的基本概念。在剖析了 IoT 的定义之后，IoT 设备将执行 4 项功能，包括传感、反应、收集和通信。所有这些功能都是通过物联网设备上使用不同的组件来完成的。

其中，传感机制通过传感器予以执行；反应或控制则由执行机构完成；收集操作则是各种传感器的功能之一；通信由提供网络连接的芯片执行。需要注意的一点是，在 IoT 中，所有这些组件均通过互联网实现访问和控制。物联网设备往往具备自身功能，当纳入至物联网生态系统后，其价值将会得到更大的发挥。

典型的物联网可以由许多物理对象组成，对象间彼此连接，并连接到一个集中的云服务器上，如图 11.1 所示。

物联网元素分布在多个层，同时存在各种各样的参考架构，可用于开发物联网系统。一般来说，一个 5 层模型即可描述物联网，包含物理对象层、设备层、网络层、服务层和应用程序层。每一层或级负责相关功能且涵盖各种组件。

1. 物理对象层

物理对象层包括人、动物、汽车、树木、冰箱、火车、工厂、家居等，事实上，任何需要被监控和控制的事物都可以连接至物联网。



图 11.1 创建的 IoT 网络（数据源自 IBM）

2. 设备层

设备层包含了诸如传感器、执行器、智能手机、智能设备和射频识别标签（RFID）等内容。其中，传感器类型也是多种多样，例如体传感器、家居传感器，以及基于工作环境类型的环境传感器。设备层是 IoT 系统的核心，其中，各种传感器用于感知真实世界的环境，包括监测温度、湿度、液体流动、化学物质、空气、压力等的传感器。通常，设备上需要配置模拟数字转换器（ADC），以便将真实世界的模拟信号转换成微处理器能理解的数字信号。

这一层的执行器提供了对外部环境的控制方法，例如启动电机或打开一扇门。此外，这一类组件还需要使用到数字-模拟转换器，以便将数字信号转换成模拟信号。当 IoT 设备控制机械部件时，这一点十分重要。

3. 网络层

网络层由各种网络设备组成，用于在设备之间提供互联网连接，以及 IoT 系统中的云或服务。此类设备包括网关、路由器、集线器和交换机，以及两种类型的通信方式。首先是水平通信方式，包括无线电、蓝牙、WiFi、以太网、LAN、ZigBee 和 PAN，并以此提供物联网设备之间的通信。其次，还可与邻接层进行沟通，通常是通过互联网，提供机器、人或上方各层之间的通信。其中，第一层可以选择性地包含在设备层中，因为其物理位置位于设备层，设备可以在同一层上彼此通信。



4. 管理层

该层为物联网系统提供管理层，其中包含了处理从物联网设备收集数据的平台，并将其转化为有意义的内容。此外，该层还涵盖了设备管理、安全管理和数据流管理；除此之外，还负责管理设备和应用程序层之间的通信。

5. 应用程序层

应用程序层包括在 IoT 网络上运行的应用程序，涉及交通、医疗、财务、保险或供应链管理，最终内容取决于具体需求。当然，实际清单并非无限延展，但大多数物联网应用可以归入这一层，如图 11.2 所示。

应用程序层 交通、金融、保险等
管理层 数据处理、分析、安全管理
网络层 LAN、WAN、PAN 和路由器
设备层 传感器、执行机构和智能设备
物理对象 人、车辆和家居等

图 11.2 IoT 中的 5 层模型

随着传感器的价格不断降低，以及硬件和带宽的升级，物联网近年来越来越受到人们欢迎，目前在医疗、保险、供应链管理、家庭自动化、工业自动化和基础设施管理等多个领域都可看到物联网的身影。此外，技术的进步（如 IPv6 的普及）、体积更小且功能更加强大的处理器，以及优质的互联网接入，也在 IoT 的推广方面起到了至关重要的作用。IoT 的优点包括节省成本、帮助企业制定重要决策，进而提升计算性能（针对 IoT 设备提供的数据）。另外还针对数百万台物联网设备的原始数据进行分析，并提供有意义的见解，可以帮助我们做出及时有效的商业决策。

常规的物联网模型一般基于集中式范例，物联网设备通常与云基础设施或中央服务器连接，以便报告和处理相关数据。这种集中化形式可能会被不法分子所利用，包括黑客攻击和数据盗窃。此外，对于某家集中式服务提供商，缺少个人数据的控制也会导致安全性和隐私方面的问题。虽然某些方法和技术可以在常规物联网模型的基础上构建安全的物联网系统，但区块链可以给物联网带来更大的好处。基于区块链的物联网模式与传统的物联



网模式不同。根据 IBM 的说法，IoT 的区块链有助于实现信任机制、降低成本和加速交易处理。此外，去中心化是区块链技术的核心内容，据此可消除物联网中的单点故障。例如，中央服务器可能无法处理数十亿物联网设备在高频率下产生的数据量。相应地，区块链提供的 P2P 通信模型则有助于降低成本——无须构建高成本的集中式数据中心，也不需要为安全问题提供复杂的公共密钥基础设施。设备之间可以直接或通过路由器进行通信。

根据研究人员和各家公司的估计，到 2020 年，将会有大约 220 亿台连接到互联网的设备。当数以亿计的设备连接到互联网时，很难想象，集中式基础设施如何在应付带宽、服务和可用性等问题的同时不会导致成本的大幅提升。相比之下，基于区块链的物联网能够在当前物联网模型中解决可扩展性、隐私和可靠性问题。

区块链使事物能够直接相互通信和交易，而智能合约的操作和金融交易也可以直接发生在设备之间，且不再需要中间人、权威机构或人工干预。例如，如果旅馆里的房间处于空闲状态，则可启动租赁业务、协商租金，并且为付以适当费用的用户开启门锁。另一个例子是，如果一台洗衣机中洗涤剂用完，即可通过网络进行订购，并根据智能合约中的编程逻辑获取最优的价格。

针对之前提到的 5 层物联网模型，可在网络层上添加区块链层以适应区块链模型。这一层将运行智能合约，并提供安全、隐私、完整性、自主性、可扩展性以及物联网系统的去中心化服务。在这种情况下，管理层只能由与分析 and 处理相关的软件组成，安全性和控制可以转移到区块链层，如图 11.3 所示。

应用程序层 交通、金融、保险等
管理层 数据处理、分析、安全管理
区块链层 安全、P2P（M2M）自主性交易、去中心化和智能合约
网络层 LAN、WAN、PAN 和路由器
设备层 传感器、执行机构和智能设备
物理对象 人、车辆和家居等

图 11.3 基于区块链的 IoT 模型



在上述模型中，其他层可能保持不变，但作为 IoT 网络所有参与者之间的一个中间件，该模型引入了一个额外的区块链层。

同时，在抽取上述各层后，该模型还可描述为 P2P 物联网，如图 11.4 所示。其中，所有设备间均可彼此通信和协商，且无须中央命令和控制实体的参与。

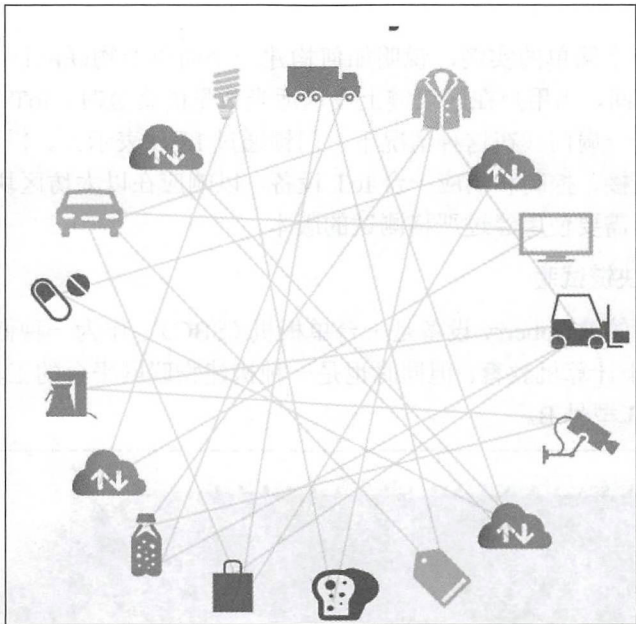


图 11.4 基于区块链的直接通信模型（内容源自 IBM）

由于区块链去中心化方案降低了设备管理难度,因而成本将会随之减少。另外,区块链还可对 IoT 网络的性能予以优化。在这种情况下,无须集中存储数百万台设备的 IoT 数据,存储和处理需求可以分发到区块链的所有 IoT 设备上。对于 IoT 数据的处理和存储,大型数据中心将不再必需。

区块链 IoT 还可以阻止拒绝服务攻击。相比较而言，黑客可以更容易地攻击集中式服务器或数据中心；但考虑到区块链的分布式和去中心化性质，这一类攻击将不复存在。此外，如果数十亿台设备连接到互联网上，那么在传统中央服务器上，管理所有这些设备的安全性和更新几乎是不可能的。针对这一问题，区块链提供了一种解决方案，即设备间以安全的方式直接通信，甚至彼此间可请求固件和安全的更新操作。在区块链网络上，这一类通信行为可安全地记录下来且不可更改，进而为系统提供可审核性、完整性和透明性。这在传统的 P2P 系统中是无法实现的。



综上所述,随着 IoT 和区块链的融合,其优势已得到明显的体现。学术界和业界对此也做出了不懈的努力,许多项目提案均采用了基于区块的物联网解决方案。例如,IBM Blue Horizon 和 IBM Bluemix 都是支持区块链的 IoT 平台;像 Filament 这样的初创公司已经提出了一种去中心化网络的新想法,该网络可以让 IoT 的设备直接和自主地通过智能合约进行交易。

下面将考察一个简单的实例,说明如何构建一个简单的物联网设备,并将其连接到以太坊区块链。期间,当用户在区块链上发送适当数量的资金时,IoT 设备将连接到以太坊区块链,并打开一扇门(在这种情况下,门锁通过 LED 表示)。需要说明的是,该示例仅演示了如何连接、控制和响应一台 IoT 设备,以响应在以太坊区块链上的某些事件。在实际产品中,则需要使用经过严格测试的版本。

6. 物联网区块链试验

当前示例所用的 Raspberry 设备是一台单板机(SBC)。作为一种低成本计算机设备,Raspberry Pi 辅助于计算机教育,但同时也是一种构建物联网平台的工具。图 11.5 所示内容为 Raspberry Pi 3 型号 B。

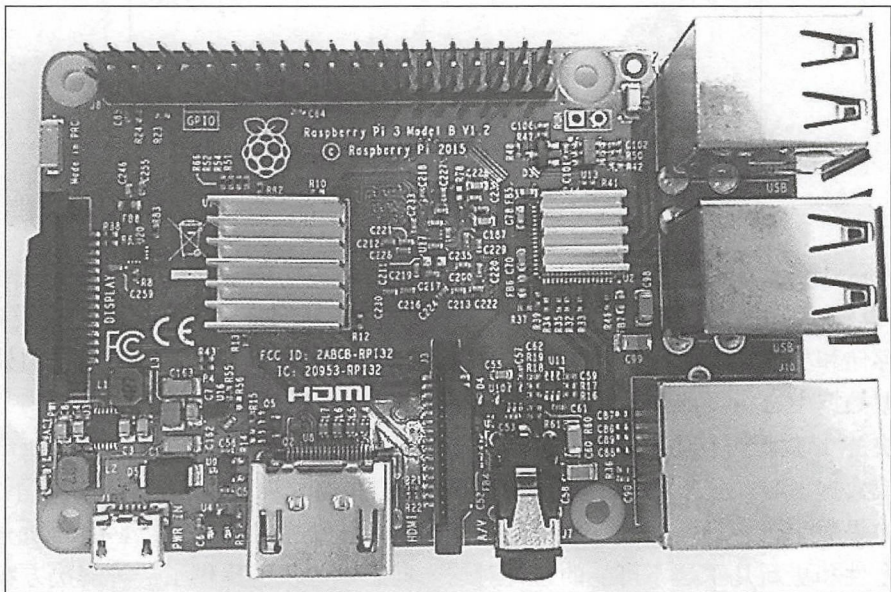


图 11.5 Raspberry Pi 3 型号 B

其中,Raspberry Pi 将用作与以太坊区块链连接的物联网设备,并执行某项操作以响应智能合约调用。

首先, 需要设置 Raspberry Pi, 该过程可通过使用 NOOBS 来实现, 它提供了一个简单的方法来安装 Raspbian 或任何其他操作系统。读者可访问 <https://www.raspberrypi.org/downloads/noobs/> 进行下载和安装。除此之外, 链接 <https://www.raspberrypi.org/downloads/raspbian/> 仅提供了 Raspbian 的安装操作。另外, 链接 <https://github.com/debian-pi/raspbian-ua-netinst> 可用于安装 Raspbian OS 的非 GUI 版本(最小化版本)。出于演示目的, NOOBS 用于安装 Raspbian。另外, 这里假设 Raspbian 均已安装了 Raspberry Pi 的 SD 内存卡。

Raspbian 操作系统安装完毕后, 下一步即是针对 Raspberry Pi ARM 平台下载 geth 二进制文件。在 Raspberry Pi Raspbian 操作系统中, 该平台可通过相关命令确认。命令输出显示操作系统当前正在运行的架构。在本例中是 armv7l, 如图 11.6 所示, 因此须下载 geth 的 ARM 兼容二进制文件。

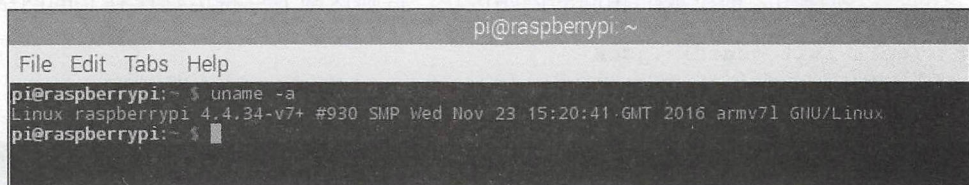


图 11.6 Raspberry Pi 架构

相关操作步骤如下:

- ❑ 下载 geth。注意, 当前示例将下载特定版本; 读者可访问 <https://geth.ethereum.org/downloads/> 下载其他版本。具体操作如下所示:

```
wget https://gethstore.blob.core.windows.net/builds/geth-linux-arm7-1.5.6-2a609af5.tar.gz
```

- ❑ 将相关内容解压至某个目录中。例如, 利用下列 tar 命令, 将自动生成名为 geth-linux-arm7-1.5.6-2a609af5 的目录:

```
tar -zxvf geth-linux-arm7-1.5.6-2a609af5.tar
```

这将创建一个名为 geth-linux-arm7-1.5.6-2a609af5 的目录, 并将 geth 二进制文件和相关文件提取到该目录中。另外, 可以将 geth 二进制文件复制到 /usr/bin, 或 Raspbian 中的相关路径下, 使其可以从操作系统的任何地方获得。当下载完成后, 下一步是创建创始区块。

第 8 章曾讲到, 须使用相同的创始区块; 创始文件可以从网络上的其他节点复制, 如图 11.7 所示。或者, 也可以生成一个全新的创始区块, 具体内容可参考第 8 章。

```
pi@raspberrypi:~/geth-linux-arm7-1.5.6-2a609af5 $ cat genesis.json
{
  "nonce": "0x00000000000000042",
  "timestamp": "0x0",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x0",
  "gasLimit": "0x4c4b40",
  "difficulty": "0x200",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "alloc": {}
}
```

图 11.7 创始文件

当 genesis.json 文件复制到 Raspberry Pi 后，可以运行下面的命令生成创始区块。需要注意的是，须使用之前生成的相同的创始区块，否则这些节点将运行在独立的网络上。

```
$ ./geth init genesis.json
```

这将生成如图 11.8 所示的结果。

```
pi@raspberrypi:~/geth-linux-arm7-1.5.6-2a609af5 $ ./geth init genesis.json
I0110 23:37:15.714795 cmd/utlils/flags.go:612] WARNING: No etherbase set and no accounts found as default
I0110 23:37:15.715283 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:37:15.794383 ethdb/database.go:176] closed db:/home/pi/.ethereum/geth/chaindata
I0110 23:37:15.794723 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:37:15.923300 core/genesis.go:93] Genesis block already in chain. Writing canonical number
I0110 23:37:15.923895 cmd/geth/chaincmd.go:131] successfully wrote genesis block and/or chain rule set: f2b2ffed01907a845a01d1dea21e5a
ec021e8e68b5ec9ffccb82df
```

图 11.8 初始化创始文件

在创始区块创建完毕后，需要将对应点添加到网络中。这可以通过创建一个名为 static-nodes.json 的文件实现，如图 11.9 所示，其中包含了 Raspberry Pi 上 geth 同步连接的、对应点的编码 ID。

```
pi@raspberrypi:~/.ethereum $ cat static-nodes.json
[
  "enode://44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f@192.168.0.19:30301"
]
```

图 11.9 静态节点配置

通过运行下列命令，可以从 geth JavaScript 控制台获得此信息，该命令应该在 Raspberry 将连接的对等点上运行。

```
> Admin.nodeInfo
```

这将生成如图 11.10 所示的结果。


```

> admin.nodeInfo
{
  enode: "enode://44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3b2787375e932fb4885885f6452f6efa77f@[:]:30301",
  id: "44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3b2787375e932fb4885885f6452f6efa77f",
}

```

图 11.10 节点信息

随后,可遵循后续指令将 Raspberry Pi 连接至私有网络上的其他节点。在当前示例中,Raspberry Pi 将连接至第 8 章创建的网络 ID 786。这里,关键之处在于,应使用之前创建的同一初始区块,以及不同的端口号。此处并未对不同端口这一条件做出严格规定。如果两个节点运行于某个私有网络,同时需要从外部环境访问网络,则可采用 DMZ/路由器和端口转发这一组合方式。因此,此处建议使用不同的 TCP 端口,以使端口转发处于正常工作状态。后续命令中的标识开关(之前未予介绍)允许为节点指定一个标识名称。

❑ 首节点设置。首先, geth 需要在首个节点上启动,并使用如下命令:

```

$ geth --datadir .ethereum/privatenet/ --networkid 786 --maxpeers 5 --rpc --rpcapi web3,eth,debug,personal,net --rpcport 9001 --rpccorsdomain "*" --port 30301 --identity "drequinox"

```

对应结果如图 11.11 所示。

```

imran@drequinox-0P7010:~$ geth --datadir .ethereum/privatenet/ --networkid 786 --maxpeers 5 --rpc --rpcapi web3,eth,debug,personal,net --rpcport 9001 --rpccorsdomain "*" --port 30301 --identity "drequinox"
I0110 23:26:46.032878 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/.ethereum/privatenet/eth/chaindata
I0110 23:26:46.072986 ethdb/database.go:176] closed db:/home/imran/.ethereum/privatenet/eth/chaindata
I0110 23:26:46.073243 node/node.go:175] instance: Geth/drequinox/v1.5.2-stable-c8695209/linux/go1.7.3
I0110 23:26:46.073258 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/.ethereum/privatenet/eth/chaindata
I0110 23:26:46.082654 eth/backend.go:193] Protocol Versions: [63 62], Network Id: 786
I0110 23:26:46.083188 core/blockchain.go:214] Last header: #7991 [999c534f...] TD=11652654509
I0110 23:26:46.083203 core/blockchain.go:215] Last block: #7991 [999c534f...] TD=11652654509
I0110 23:26:46.083210 core/blockchain.go:216] Fast block: #7991 [999c534f...] TD=11652654509
I0110 23:26:46.083929 p2p/server.go:336] Starting Server
I0110 23:26:48.239776 p2p/discover/udp.go:217] Listening, enode://44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3b2787375e932fb4885885f6452f6efa77f@[:]:30301
I0110 23:26:48.239893 p2p/server.go:604] Listening on [::]:30301
I0110 23:26:48.240913 node/node.go:340] IPC endpoint opened: /home/imran/.ethereum/privatenet/eth.ipc
I0110 23:26:48.241212 node/node.go:410] HTTP endpoint opened: http://localhost:9001
I0110 23:42:58.206205 eth/backend.go:479] Automatic pregeneration of ethash DAG ON (ethash dir: /home/imran/.ethash)
I0110 23:42:58.206217 miner/miner.go:136] Starting mining operation (CPU=8 TOT=9)

```

图 11.11 首节点上的 geth

一旦 geth 启动后,则应处于运行状态;此时,另一个 geth 实例应在 Raspberry Pi 节点上启动。

❑ Raspberry Pi 节点设置。在 Raspberry Pi 上,需要运行下列命令以启动 geth,并将其与其他节点同步(当前示例中仅存在一个节点)。

```
$ ./geth --networkid 786 --maxpeers 5 --rpc --rpcapi
web3,eth,debug,personal,net --rpccorsdomain "*" --port 30302 -
identity "raspberrry"
```

这将生成如图 11.12 所示的结果。当输出结果显示“Block synchronization started”消息后，即表明节点已与其对等点成功连接。

```
pi@raspberrypi:~/geth-linux-arm7-1.5.6-2a609af5 $ ./geth --networkid 786 --maxpeers 5 --rpc --rpcapi web3,eth,debug,personal,net --
--rpccorsdomain "*" --port 30302 --identity "raspberrry"
I0110 23:38:04.654374 cmd/utils/flags.go:612] WARNING: No etherbase set and no accounts found as default
I0110 23:38:04.654776 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:38:04.693111 ethdb/database.go:176] closed db:/home/pi/.ethereum/geth/chaindata
I0110 23:38:04.696937 node/node.go:176] instance: Geth/raspberrry/v1.5.6-stable-2a609af5/linux/go1.7.4
I0110 23:38:04.697042 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:38:04.847835 eth/backend.go:191] Protocol Versions: [63 62], Network Id: 786
I0110 23:38:04.849753 eth/backend.go:219] Chain config: {ChainID: 0 Homestead: <nil> DAO: <nil> DAOsupport: false EIP150: <nil> EIP1
P158: <nil>}
I0110 23:38:04.857847 core/blockchain.go:216] Last header: #2668 [6776ef24...] TD=708187563
I0110 23:38:04.858174 core/blockchain.go:217] Last block: #2668 [6776ef24...] TD=708187563
I0110 23:38:04.858349 core/blockchain.go:218] Fast block: #2668 [6776ef24...] TD=708187563
I0110 23:38:04.866705 p2p/server.go:340] Starting Server
I0110 23:38:10.223170 p2p/discover/udp.go:227] Listening, enode://98ba36ecea7ff0f11803d634da45752abd25101f20a62f23427afc3f280017bc134
b195aced69c3b01c223f1d638a52697a1bbbf967fc04274086.15.44.209:30302
I0110 23:38:10.224031 p2p/server.go:608] Listening on [::]:30302
I0110 23:38:10.233788 node/node.go:341] IPC endpoint opened: /home/pi/.ethereum/geth.ipc
I0110 23:38:10.237027 node/node.go:411] HTTP endpoint opened: http://localhost:9002
I0110 23:38:20.225637 eth/downloader/downloader.go:326] Block synchronisation started
I0110 23:38:49.583631 core/blockchain.go:1067] imported 1 blocks, 0 txs ( 0.000 Mg/s) in 14.018s ( 0.000 Mg/s). #2669 [76077955
I0110 23:38:49.622191 core/blockchain.go:1067] imported 5 blocks, 0 txs ( 0.000 Mg/s) in 38.520ms ( 0.000 Mg/s). #2674 [76077955
```

图 11.12 Raspberry Pi 上的 geth

这可通过在两个节点的 geth 控制台上运行如图 11.13 所示的命令来进一步验证。另外，geth 可在 Raspberry Pi 上运行下列命令进行连接：

```
$ geth attach
```

```
> admin.peers
[[{
  caps: ["eth/62", "eth/63"],
  id: "44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e932f
c4085885f5452f6efa77f",
  name: "Geth/drequisinox/v1.5.2-stable-c8695209/linux/go1.7.3",
  network: {
    localAddress: "192.168.0.21:56550",
    remoteAddress: "192.168.0.19:30301"
  },
  protocols: {
    eth: {
      difficulty:
        head: "0x2d32c90b4c9dacea9a109b0ae52c1ebf511915bb618a2d3c55a80a63852e89f6",
      version:
    }
  }
}]
```

图 11.13 运行于 Raspberry Pi 上的 geth 控制台管理对等点

类似地，在首节点上，可运行下列命令连接 geth：

```
$ geth attach ipc:.ethereum/privatenet/geth.ipc
```

一旦控制台可用，可以运行 admin.peers 显示其他连接节点的详细信息，如图 11.14 所示。


```
> admin.peers
[[
  caps: ["eth/62", "eth/63"],
  id: "98ba36ecea7ff011803d034da45752abd25101f20a62f23427afc3f280017bc134833dd5ba400bb195ac6ed59c3b01
ca2a3f14638a52697a1bb1bf967fc84274",
  name: "Geth/raspberry/v1.5.6-stable-2a609af5/linux/go1.7.4",
  network: {
    localAddress: "192.168.0.19:30301",
    remoteAddress: "192.168.0.21:56512"
  },
  protocols: {
    eth: {
      difficulty:
      head: "0x1188f58b4900a1d771d333141ea9400d78400bb8e561494ab436519ae64e1e34",
      version:
    }
  }
}]
```

图 11.14 运行于其他对等点上的 geth 控制台的 admin.peers 命令

一旦两个节点均处于运行状态，即可安装其他内容，以设置当前试验环境。具体而言，当前示例须安装 Node.js 和 JavaScript 库。首先，在 Raspberry Pi Raspbian 操作系统上需要更新 Node.js 和 npm，相关步骤如下：

□ 通过下列命令在 Raspberry Pi 上安装最新版本的 Node.js：

```
$ curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash-
```

对应结果如图 11.15 所示。鉴于输出结果占据了较大的篇幅，因而将通过多幅图像显示。

```
pi@raspberrypi:~/testled $ curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
## Installing the NodeSource Node.js v7.x repo...

## Populating apt-get cache...

+ apt-get update
Get:1 http://archive.raspberrypi.org jessie InRelease [22.9 kB]
Get:2 http://mirrors.digitalocean.com/raspbian org jessie InRelease [14.9 kB]
```

图 11.15 安装 Node.js

□ 通过 apt-get 执行更新操作。

```
$ sudo apt-get install nodejs
```

可以运行如图 11.16 所示的命令执行验证操作，以确保 Node.js 版本的正确性，以及是否安装了 npm。

```
pi@raspberrypi:~/testled $ npm -v
4.0.5
pi@raspberrypi:~/testled $ node -v
v7.4.0
pi@raspberrypi:~/testled $
```

图 11.16 验证 npm 和节点的安装

需要说明的是，图 11.16 所示的版本并非必需，npm 和 node 的最新版本均可正常工作。本章示例则采用了 npm 4.0.5 和 node v7.4.0。

- ❑ 安装以太坊 web3 npm，且需要启动 JavaScript 代码访问区块链，如图 11.17 所示。

```
pi@raspberrypi:~/testled $ npm install web3
testled@1.0.0 /home/pi/testled
└─ web3@0.18.0
   └─ bignumber.js@2.0.7 (git+https://github.com/debris/bignumber.js.git#94d7146671b9719e00a09c29b01a691bc85048c2)

npm WARN testled@1.0.0 No repository field.
pi@raspberrypi:~/testled $
```

图 11.17 安装 web3

- ❑ 类似地，还需要安装 npm install onoff，进而与 Raspberry Pi 通信，并控制 GPIO，如图 11.18 所示。

```
pi@raspberrypi:~/testled $ npm install onoff --save
testled@1.0.0 /home/pi/testled
└─ onoff@1.1.1

npm WARN testled@1.0.0 No repository field.
pi@raspberrypi:~/testled $
```

图 11.18 安装 onoff

待全部内容安装完毕后，即可执行硬件设置。对此，可采用试验电路板以及某些元件搭建电路。

相关元件如下所示。

- ❑ LED：发光二极管的简称，可用于某个事件的视觉效果显示。
- ❑ 电阻：这里需要使用到一个 330Ω 的电阻元件，并提供基于额定功率的电流。读者无须理解背后的理论知识，任何一本电子工程教材均会对此予以详细的介绍。
- ❑ 面包板（breadboard）：提供了电路的搭建方式（无须焊接）。
- ❑ T 形扩展板：可插入到面包板上，并提供了 Raspberry Pi 所有 GPIO（通用 I/O）引脚的标记视图。
- ❑ 带状电缆连接器：通过 GPIO，在 Raspberry Pi 和电路板之间提供连接。全部元件如图 11.19 所示。

如图 11.20 所示，LED 的正极（长引脚）连接到 GPIO 的第 21 针，负极（短引脚）连接到电阻器，然后再连接到 GPIO 的接地引脚（GND）。一旦建立了连接，就可以使用带状电缆简单地连接到 Raspberry Pi 上的 GPIO 连接器。

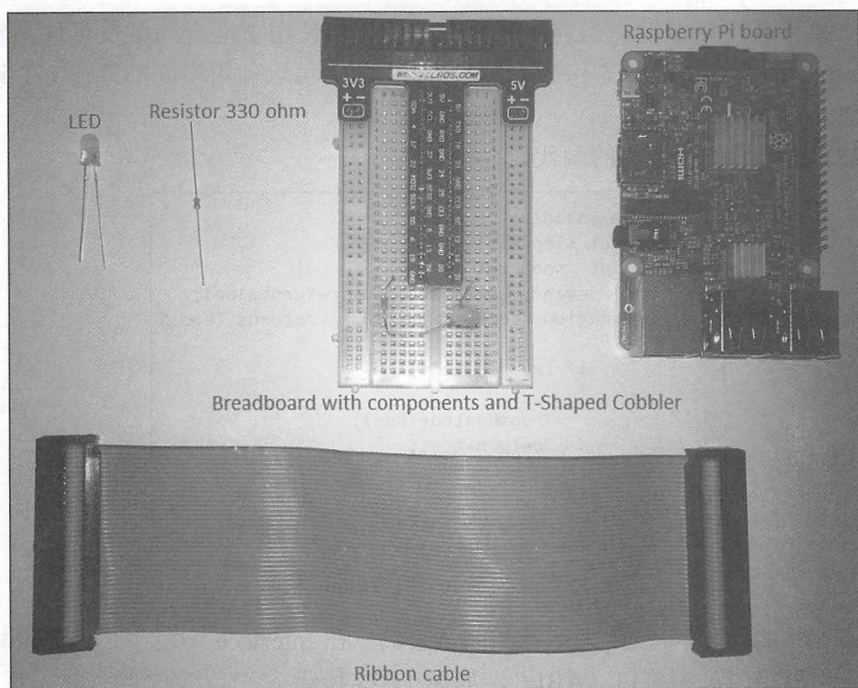


图 11.19 所需元件

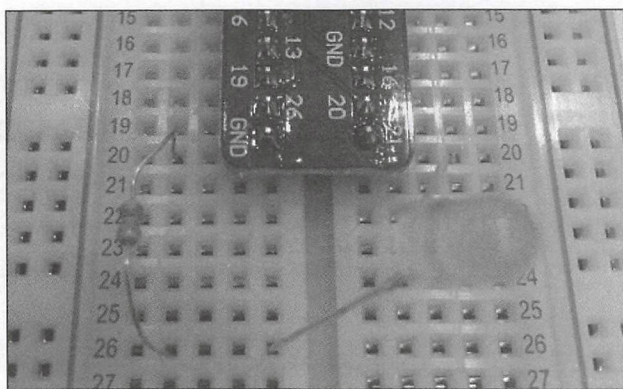


图 11.20 面包板上的元件连接

当正确地设置了连接后，Raspberry Pi 也随之被更新（包括库和 geth）。下一步就是编写简单的智能合约，如传递值与期望值不匹配，则不会触发事件；否则，如果传递值与正确值匹配，则可以通过 node.js 运行的客户端 JavaScript 程序读取事件触发器。当然，Solidity 合约可能非常复杂，同时还可处理发送给它的 Ether，如果 Ether 的数量等于所需

数量，那么事件就可以触发。当前示例旨在演示如何使用智能合约触发事件，然后由运行在 Node.js 上的 JavaScript 程序读取这些事件；随后可以使用不同的库在 IoT 设备上触发相关操作。

图 11.21 显示了智能合约的源代码。

```
1 pragma solidity ^0.4.0;
2 contract simpleIoT {
3     uint roomrent=10;
4     event roomRented(bool returnValue);
5     function getRent (uint8 x) returns (bool)
6     {
7         if (x==roomrent)
8         {
9             roomRented(true);
10            return true;
11        }
12    }
13 }
```

图 11.21 简单 IoT 的 Solidity 代码

Solidity 在线编译器可以用来运行和测试合约。在 Interface 字段中，可使用与合约交互所需的应用程序二进制接口（ABI），如图 11.22 所示。

The screenshot displays the Solidity online compiler interface for a contract named 'simpleIoT'. The top right corner indicates the contract size as '274 bytes'. Below the contract name, there are two buttons: 'At Address' and 'Create'. The interface is divided into four main sections: 'Bytecode', 'Interface', 'Web3 deploy', and 'Metadata location'. The 'Bytecode' section shows a long hexadecimal string. The 'Interface' section displays the ABI in JSON format. The 'Web3 deploy' section contains a JavaScript code snippet for deploying the contract using web3.js. The 'Metadata location' section shows a BZZ address for the contract's metadata.

```
var simpleiotContract = web3.eth.contract([{"constant":false,"inputs":[{"name":"x","type":"uint8"}],"name":"getRent","outputs":[{"name":"returnValue","type":"bool"}]}]);
var simpleiot = simpleiotContract.new({
  from: web3.eth.accounts[0],
  data: '0x60604052600a60005534610000575b60f58061001d6000396000f30060606040526000357c010c',
  gas: '4700000'
}, function (e, contract) {
  console.log(e, contract);
  if (typeof contract.address !== 'undefined') {
    console.log('Contract mined! address: ' + contract.address + ' transaction hash: ' + tx.hash);
  }
});
```

图 11.22 Solidity 在线编译器

Raspberry 节点可采用两种方法，并通过 Web3 接口连接到私有区块链。首先，树莓设备运行自己的 `geth`，并维护自己的账本。但在资源受限的设备中，设备无法运行完整的 `geth` 节点；某些时候甚至无法运行一个轻量级节点。在这种情况下，Web3 提供商可初始化并连接到适当的 RPC 通道，稍后将在客户端 JavaScript Node.js 程序列表中显示这一点。这两种方法的比较如图 11.23 和图 11.24 所示。

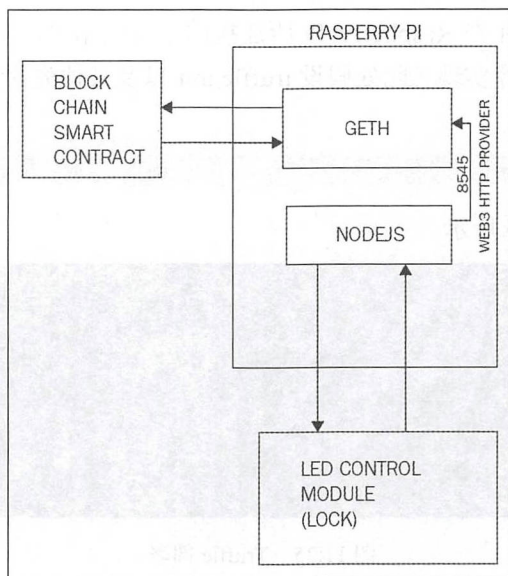


图 11.23 房屋租赁 IoT（包含本地账本的 IoT）应用程序的架构

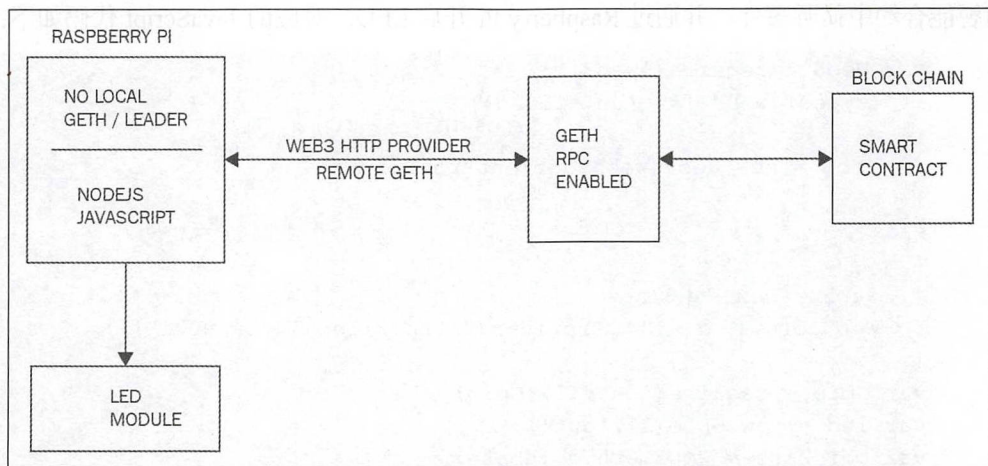


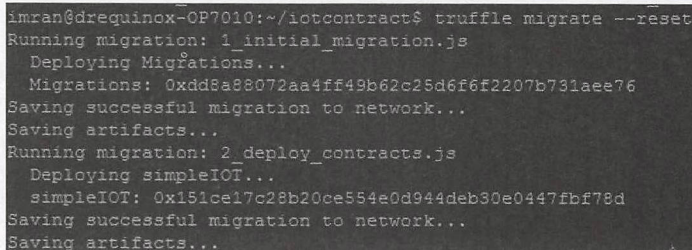
图 11.24 房屋租赁 IoT（未包含本地账本的 IoT）应用程序的架构

公开显示 RPC 接口将导致某些安全问题，因此建议当前选择方案仅可用于私有网络中。如果需要使用公共网络，则需要辅以适当的安全措施。例如，仅允许已知的 IP 地址连接到 geth RPC 接口。对此，可通过禁用对等点发现机制和 HTTP-RPC 服务器监听接口来实现。读者可参考 geth 帮助文档以了解更多信息。除此之外，还可采用传统的网络安全措施，如防火墙、传输层安全（TLS）和证书等。

目前，Truffle 可用于在 Raspberry Pi 所连接的私有网络 ID 786 上部署合约。Truffle 部署过程可通过下列命令实现（此处假设 truffle init 以及其他先决条件均已设置完毕，具体内容可参考第 8 章）：

```
$ truffle migrate
```

对应结果如图 11.25 所示。



```
imran@drequinox-OP7010:~/iotcontract$ truffle migrate --reset
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0xdd8a88072aa4ff49b62c25d6f6f2207b731aee76
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying simpleIOT...
  simpleIOT: 0x151ce17c28b20ce554e0d944deb30e0447fbf78d
Saving successful migration to network...
Saving artifacts...
```

图 11.25 Truffle 部署

一旦正确部署了合同，即可编写 JavaScript 代码、通过 Web3 连接到区块链、从区块链的智能合约中侦听事件，并通过 Raspberry Pi 开启 LED。对应的 JavaScript 代码如下：

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined')
{
    web3 = new Web3(web3.currentProvider);
}
else
{
    web3 = new Web3(new
    Web3.providers.HttpProvider("http://localhost:9002"));
}
var Gpio = require('onoff').Gpio;
var led = new Gpio(21, 'out');
var coinbase = web3.eth.coinbase;
var ABIString = '[{"constant":false,"inputs":
```



```
[{"name":"x","type":"uint8"}],{"name":"getRent","outputs":[{"name":"","type":"bool"}],{"payable":false,"type":"function"},{"anonymous":false,"inputs":[{"indexed":false,"name":"returnValue","type":"bool"}],{"name":"roomRented","type":"event"}]';
var ABI = JSON.parse(ABIString);
var ContractAddress = '0x151ce17c28b20ce554e0d944deb30e0447fbf78d';
web3.eth.defaultAccount = web3.eth.accounts[0];
var simpleiot = web3.eth.contract(ABI).at(ContractAddress);
var event = simpleiot.roomRented({}, function(error, result) {
  if (!error)
  {
    led.writeSync(1);
  }
});
```

需要注意的是,在当前示例中,合约地址 0x151ce17c28b20ce554e0d944deb30e0447fbf78d 与当前部署相关,读者在运行该示例时,地址将有所变化。在合约部署后,可简单地在文件中将地址修改为对应结果。此处,JavaScript 代码可置于 Raspberry PI 上的某个文件中,例如 index.js,并通过下列命令运行:

```
$ sudo nodejs index.js
```

这将启动当前程序,且运行于 Node.js,同时监听源自智能合约中的各种事件。当程序正确运行时,智能合约可通过 Truffle 控制台进行调用,如图 11.26 所示。

```
imran@drequinox-OP7010:~/iotcontract$ truffle console
truffle(default)> simpleIOT.deployed().getRent(10)
'0x7e8b33f5354a73e2874ef29b26ea89d5811f23978778a9c05e11d5b19cd0fd40'
```

图 11.26 与合约进行交互

在当前示例中, .getRen 函数将被调用,对应的期望参数值为 10。

合同被挖掘后,将触发 roomRented,并开启 LED 灯。在本例中,当前设备仅是简单的 LED,但也可以是任何物理设备,例如,可以通过执行器控制的房间锁。如果一切正常,LED 将作为智能合约函数调用的结果被打开,如图 11.27 所示。

在当前示例中,可构建一个由 IoT 设备组成的私有网络,在每个节点上运行 geth 客户端,侦听来自智能合约中的事件,并据此触发相应的操作。该示例的目的很简单,并演示了以太坊网络的基本原理,其构建方式包括物联网设备以及智能合约驱动的物理设备控制行为。

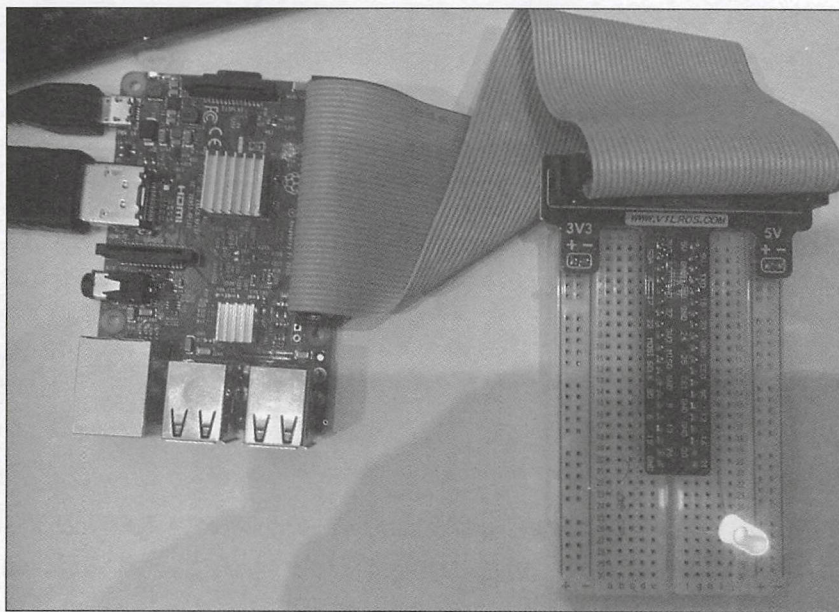


图 11.27 包含 LED 元件的 Raspberry Pi

11.2 政府机构

区块链应用也可体现在政府职能方面，并将目前的电子政务模式提升一个层次。首先，本节将探讨电子政务的一些背景，随后以此考察电子投票、国土安全（边境管制）和电子身份证（公民身份证）等一些用例。

作为范例，电子政务利用资讯和通信技术向市民提供公共服务。这一概念并不新鲜，且在世界各国都已予以实施，但区块链开辟了一条新的探索之路。许多政府正在研究使用区块链技术来管理和提供公共服务。透明度、可审核性和完整性是区块链的属性，可以有效地管理各种政府职能。

11.2.1 边境管理

为了有效地打击非法入境、恐怖主义以及人口贩运，几十年来，边境自动控制系统一直在不遗余力地发挥着自身的作用。

机器可读的旅行证件，特别是生物识别护照，均应用于自动边境控制系统中；然而，目前的系统在一定程度上依然受到了各种限制。对此，区块链技术可以提供相应的解决

方案。MRTD（机器可读的旅行证件）标准是由国际民用航空组织（ICAO）在 ICAO 9303 文件中制定的，并已被世界上许多国家所采纳。其中，每份护照都配置了各种各样的安全和身份属性，可以用来识别护照持有者，从而严厉打击了伪造护照这一类不法行为。相关属性包括生物特征，如视网膜扫描、指纹、面部识别，以及标准 ICAO 所指定的各项功能，例如机器可读区域（MRZ），以及在护照首页可见的其他文本属性。

当前边界管理系统的一个关键点是集中化问题。具体而言，系统由单一实体控制，而执法机构之间难以实现数据共享，从而使得追踪嫌疑人变得较为困难。另一个问题则与旅行证件黑名单的即时执行有关，例如实时追踪和控制可疑的旅行证件。目前，尚不存在任何机制可即时将护照列入黑名单或撤销。

针对上述问题，区块链提供了一种解决方案，即在智能合约中维护一个黑名单，并可以根据需要予以更新，任何变化都可以立即被所有机构和边境控制点看到，进而对可疑旅行证件进行实时监控。当然，像 PKI 和 P2P 网络这一类传统机制也适用于此，但区块链的优点更为突出。区块链可以简化整个系统，无须复杂的网络和 PKI 设置，同时还有效地降低成本。此外，区块链系统还以加密方式提供了不可变性特征，这有助于实施审计操作，并阻止任何欺诈活动。

考虑到扩展问题，所有旅行证件的完整数据库可能无法存储在区块链上，但是后端分布式数据库（如 BigChainDB、IPFS 或 Swarm）则可用于此目的。在这种情况下，带有个人生物特征的旅行证件的哈希值已存储在一个简单的智能合约中，随后可通过证件的散列值引用分布式文件系统（如 IPFS）上可用的详细数据。这样，当旅行证件被列入黑名单时，此类信息即可立即在整个分布式账本中查看，同时还可保证其真实性和完整性。这一功能也可以为反恐活动提供有效的支持，并在政府的国土安全职能中发挥至关重要的作用。

在 Solidity 中，合约可以定义一个数组存储身份和相关的生物特征记录。该数组可用于存储与护照相关的标识信息，对应标识可以是护照或旅行证件机器可读区域（MRZ）的哈希值，并与 RFID 芯片上的生物特征记录连接在一起。相应地，列于黑名单上的护照可通过一个简单的布尔字段加以识别。当通过这一初始测试后，生物验证过程可通过常规系统执行。若护照持有者可安全入境，该信息将反馈至区块链中，以供网络上的所有参与者共享。

图 11.28 显示了基于区块链边界控制系统的高级可视化方法。其中，护照可以输入至 RFID 和扫描仪中，从可读取数据页面提取可加工的信息，以及存储在 RFID 芯片中的生物特征数据。在这一阶段，还将获取护照持有人的实时照片和视网膜扫描结果。然后，这一类信息将传递到区块链，智能合约负责验证旅行证件的合法性。对此，智能合约首

先检查黑名单护照列表，然后从后端 IPFS 数据库中请求其他数据进行比较。注意，像照片或视网膜扫描这一类生物特征数据并不存储在区块链上，仅后台 (IPFS 或 BigChainDB) 数据的引用存储在区块链中。

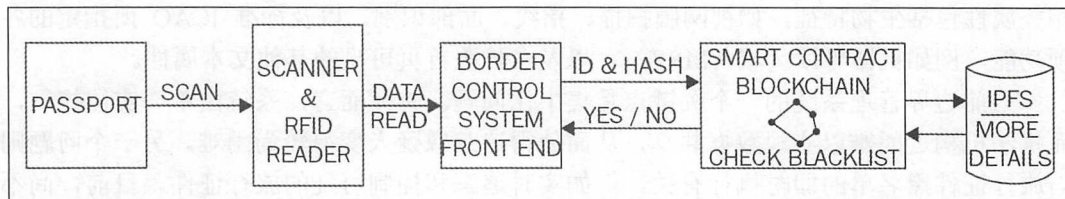


图 11.28 基于区块链的自动入境控制系统

如果护照上的数据与在 IPFS 文件或 BigChainDB 中持有的文件相匹配，并通过智能合约的逻辑验证，则护照持有者即可安全入境。

随后，信息验证结果将在区块链中传播，边界控制区块链的所有参与者可对此进行实时监控。这里，参与者可以是世界各国的国土安全部门。

11.2.2 选票机制

随着时间的推移，虽然投票过程已形成了一种较为成熟和安全的机制，但其中仍然存在某些局限性需要解决，从而达到期望的成熟水平。目前，投票系统的局限性主要体现在舞弊、操作过程中的漏洞，尤其是透明度等问题上。多年以来，人们已经构建了相对安全的投票机制，利用专门的投票设备保证安全和隐私。尽管如此，相关设备仍然存在某些可能被利用的漏洞，从而破坏设备的安全性。此类问题可能会对整个投票过程产生严重影响，并可能导致公众对政府的不信任。

以区块链为基础的投票系统通过安全性和透明度解决了上述问题。区块链使用公钥密码作为标准，并以完整性和真实性等形式提供了安全性保障。此外，区块链彰显的不变性特征保证了选票不会再次被投下。具体来讲，可以通过生物特征和维护选票列表的智能合约这一组合方式加以实现。例如，智能合约可以维护一个包含生物特征 ID（例如指纹）的选票列表，以此检测、防止重复投票问题。其次，还可利用区块链上的零知识证明保护区块链的选民隐私。

11.2.3 身份证

目前，世界各国大都推出了电子身份证。电子身份证具有诸多安全特性，例如可防

止复制或篡改等行为。然而，随着区块链技术的出现，电子身份证仍存在改进的空间。

数字身份不仅限于政府发行的电子身份证，还适用于在线社交网络和论坛。其中，多种身份可用于不同用途。基于区块的在线数字身份可对个人信息共享加以控制，用户可查看数据的使用者及其目的，并且可以控制访问权限。对于目前集中控制的基础设施，此类操作几乎不可能实现。除此之外，通过独立的政务区块链，政府发布单一身份可轻松、透明地享受多种服务。也就是说，区块链作为一个平台，政府可在此基础上提供各种服务，如养老金、税收或福利，并通过唯一 ID 访问这些服务。在本例中，区块链提供了一项不可变记录，其中涉及数字 ID 所实现的所有更改和交易，从而确保了系统的完整性和透明性。此外，公民还可对出生证明、婚姻、契约，以及区块链上的许多其他文件进行公证，以作为其存在的凭据。

目前，身份计划在各个国家均有成功的案例，有一种观点认为，或许区块链在身份管理系统中并不真正需要。虽然存在隐私性和身份信息控制等优点，但考虑到区块链技术目前尚不成熟，因而尚无法贸然纳入现实世界的身份系统。然而，各国政府正在展开积极的研究工作，并探讨区块链用于身份管理的应用方式。

此外，鉴于区块链不可改变的性质，诸如“被遗忘权”等法律条文也很难纳入区块链中。

11.2.4 其他领域

可实施区块链技术的其他政府职能还包括税收征缴、福利管理和支付、土地所有权记录管理、事件登记（婚姻、出生等）、机动车登记和执照管理。当然，实际内容并不仅限于此。随着时间的推移，政府的许多职能和流程都适用于区块模型。区块链的主要优点，如不变性、透明度和去中心化特征，可为大多数传统的政府系统带来有效的改观。

11.3 保健事业

医疗保健业被视为另一个可以通过区块链技术获益的主要行业。区块链提供了一个不可变的、可审计的、透明的系统，传统的 P2P 网络则无法很好地实现。此外，与传统的复杂 PKI 网络相比，区块链提供了一种性价比更高、更简单的基础设施。在医疗保健领域，诸如隐私、数据泄露、高成本和欺诈等重大问题通常源于互操作性、高效的流程、透明度、可审核性和有效管控的缺失。另一个棘手的问题则是假冒药品，特别是在发展中国家，这一问题引发了人们的广泛关注。

随着区块链在医疗保健行业的普及，其优势也体现得越发明显，包括成本节省、提升信任度、快速索赔处理、高可用性、消除人为失误（往往由复杂的操作程序所引起），以及防止假冒药品泛滥。

从另一个角度来看，区块链以数字货币作为挖掘奖励机制，因而蕴含了某种科学问题处理能力，进而可以帮助寻找某些疾病的治疗方案。例如 FoldingCoin，它采用 FLDC 代币奖励其矿工，分享计算机处理能力，以解决需要大计算量的科学问题。关于 FoldingCoin，读者可访问 <http://foldingcoin.net/> 获取更多信息。另一个类似的项目称作 CureCoin，对应网址为 <https://www.curecoin.net/>。目前，此类项目尚未取得实际成果，但其理念仍值得关注。

11.4 金融行业

区块链在金融行业有大量应用。在金融业中，区块链是目前这一领域中最热门的话题，一些主要的银行和金融机构均在研究如何适应区块链技术，其成本节约方面所蕴含的潜力极具吸引力。

11.4.1 保险行业

在保险行业，区块链技术可有效地防止欺诈索赔，提高索赔处理速度，并提升操作的透明度。可以想象，对于企业间的索赔处理，保险公司间的共享账本可提供快速有效的操作机制。不仅如此，随着物联网和区块链的融合，还可打造一个智能设备生态圈，所有事物均可通过区块链智能合约协商、管理自身的保险策略。

区块链可以降低索赔处理所需的全部成本和工作量。具体而言，索赔行为可以通过智能合约和保险保单持有人的相关身份自动验证和支付。例如，在智能合约、Oracle 以及物联网的辅助下，可确保事故发生时记录相关的遥感数据，并基于此类信息支付赔偿款。在智能合约评估付款条款并发现不符合相关条件后，还可即时终止支付。例如，车辆在没有经过授权的车间进行维修，或者在指定区域外使用，等等。智能合约评估索赔时涉及多项条款，具体规则一般取决于保险公司；但此处的理念是，与物联网和 Oracle 结合的智能合约可实现汽车保险行业的自动化操作。

像 Dynamis 这样的几家初创公司已经提出了基于智能合约的 P2P 保险平台，这些平台运行在 Ethereum 区块链上。这一提议最初用于失业保险，且该模型中不再涉及保险公司。读者可访问 <http://dynamisapp.com/> 以了解更多信息。

11.4.2 交易后的结算

这也是区块链技术最受欢迎的应用。目前，针对成本高昂且耗时的结算过程，多家金融机构正在尝试使用区块链技术，以实现简化、自动化处理，并提升整体流程的处理速度。

为了更好地理解这一问题，下面简要介绍交易的生命周期。交易生命周期包含 3 个步骤：执行、清算和结算。执行过程涉及两方之间的交易承诺，通过交易部门的订单管理终端或交易所进入系统流程。下一个步骤则是清算过程，即根据价格和数量等特定属性在买卖双方进行确认。在此阶段，还需验证支付账户。最后，结算过程是指买卖双方的支付交易。

在传统的交易生命周期模型中，为了便于双方之间承担信用风险，需要建立一个中央结算所。不难发现，当前方案稍显复杂——买卖双方在交易过程中只能采取这一复杂的流程，其中涉及不同的公司、经纪人、清算机构和托管人。当采用区块链技术时，包含智能合约的单一分布式账本可简化全部流程，并能使买卖双方直接对话。

特别地，交易后的结算过程一般需要 2~3 天完成，这一时长还取决于清算所和对账系统。利用共享账本方案时，区块链的所有参与者均可即时查看当前唯一的交易状态。除此之外，依托于 P2P 方案，还可降低复杂性、成本、风险，以及处理交易所需的时间。最后，通过在区块链上使用相应的智能合约，可以完全消除中介机构。

11.4.3 防范金融犯罪

了解客户（KYC）和反洗钱（AML）是预防金融犯罪的关键因素。在 KYC 案例中，目前每家机构都会维护一个客户数据副本，并通过中心化的数据提供者执行验证。该过程可能颇为耗时，并导致客户端产生延迟。针对于此，区块链提供了一种解决方案，即共享所有金融机构之间的分布式账本（其中包含了客户的确切验证身份）。其中，分布式账本只能通过参与者之间的共识机制予以更新，从而提供透明度和可审核性。这不仅可以节约成本，而且还能以更好、更为一致的方式满足相关规章制度所涉及的各项要求。

对于 AML，鉴于区块链的不可变、共享和透明特征，监管机构可以很容易地获得对私有区块链的访问，从而获取相关的监管报告数据。同时，这也将减少与当前监管报告形式相关的复杂性和成本，此类数据来自不同的系统或遗留系统，经合并、格式化后获得最终的报告数据。

区块链可以提供系统中所有金融交易的单一共享视图，且具备加密安全性、可信度以及可审计性，从而降低当前管理报告方法中所包含的成本和复杂度。

11.5 媒体行业

媒体行业的关键问题一般涉及发行、版权管理以及作者的版税支付。例如，数字音乐可以在没有任何限制的情况下被复制多次，而且复制保护的各种尝试都以某种方式被黑客攻击过。音乐家或歌曲作者的作品发行缺乏有效的控制，且可被无限制地复制多次，因此对版税支付产生严重的影响。而且，支付并不总是能够得到保证。针对版权保护和版税问题，可将消费者、艺术家和业界人士联合起来，从而实现全部流程的透明性管控。对此，区块链可以提供网络，其中，数字音乐经加密后确保只能被付费用户所拥有，该支付机制是由智能合约而不是中心媒体机构或相关部门所控制的。支付行为将根据智能合约中嵌入的逻辑和“下载”的数量自动完成。此外，非法复制的数字音乐文件将不复存在，因为所有内容均已被记录在案，并且在区块链上以透明、不可变的方式存在。例如，音乐文件可与所有者信息和时间戳一同存储，并在区块链网络中予以跟踪。此外，拥有合法副本的消费者会以加密方式与内容绑定；除非经所有者授权许可，否则无法分发至另一个所有者。一旦所有数字内容在区块链上被记录下来，版权和转让就可以轻松地通过区块链进行管理。随后，智能合约可以控制所有相关方的发行和支付行为。

11.6 本章小结

区块链技术的应用涉及多种行业，并为现有的解决方案带来了较大的改观。本章讨论了可以从区块链技术获益的 5 个主要行业。首先讨论的是另一项革命性技术——物联网。物联网与区块链的结合，可以突破现有限制，同时给物联网产业带来了巨大的利益。本章在物联网方面投入了较大的篇幅，因为二者间可实现完美的结合。针对基于区块链的物联网服务，已产生了 PaaS 形式的应用示例和平台，如 IBM Watson IoT 区块链。另外，IBM Blue Horizon 目前正处于尝试阶段，同样是一个去中心化的区块链物联网网络。其次，本章讨论了区块链在政府部门中的应用，相关流程可通过透明、安全、可靠的方式执行，包括国土安全、身份证和福利支出等服务。此外，本章还讨论了金融领域方面的问题，并探讨了基于区块链技术的解决方案。区块链在金融领域的应用仍处于探索阶段，尚未形成以区块链为基础的应用系统。最后，本章讨论了医疗健康和音乐产业方面的内容，其改进措施均建立在区块链技术核心属性的基础之上，例如去中心化、透明性、可靠性和安全性。然而，在区块链技术投入使用之前，还需要解决某些挑战性问题，第 12 章将对此加以讨论。

第 12 章 可扩展性和其他挑战

本章讨论了区块链在成为主流技术之前所面临的一些挑战性问题。虽然目前已出现了各种用例和概念证明系统，且均工作良好，但仍有必要解决区块链中的某些限制条件，以进一步提升其适用性。

其中，可扩展性和隐私性可视为首要问题，同时也是需要解决的较为重要的制约条件，尤其是对隐私性要求较高的领域。这两个问题正成为阻碍区块链技术发展的主要因素。针对于此，本章将介绍相关研究成果。除了隐私和可扩展问题之外，其他挑战还包括规则、集成、适应性和安全性。尽管比特币区块链中的安全问题已然可提供可靠的保护，并且经受住了时间的考验，但在某些特定场合下，仍需对安全问题重新审视。

12.1 可扩展性

在过去的几年中，这一问题一直是辩论、研究和媒体关注的焦点。可扩展性体现了区块链广泛的适应性与有限的私人使用之间的差异。该领域涌现出了大量的研究成果，以及相关的解决方案，稍后将对此加以讨论。

从理论上讲，解决可扩展性问题的一般方法通常都是围绕着增加协议级别进行的。例如，比特币可扩展性的常见解决方案是增加其区块大小。其他建议还包括某些链外处理方案，也就是说，将特定处理卸载到链外网络，如链外状态网络。根据上面提到的解决方案，具体实施方案一般可分为两类：首先是链内方案，即修改区块链所操作的基础协议；其次是链外方案，即利用网络并处理链外资源，进而增强该区块链。

关于如何处理区块链中的各项制约条件，Miller 在其论文“扩展去中心化区块链”中还提出了其他方案。Miller 指出，区块链可分为各种抽象层，称为平面（plane），各平面负责执行特定的功能，包括网络平面、共识平面、存储平面、视图平面以及侧平面。这种抽象机制使得瓶颈和限制在各个平面内以结构方式被各自解决。后续内容将采用比特币系统作为参考并对各层进行简要介绍。

下面首先讨论网络平面。如前所述，网络平面的功能是传播交易；在比特币中，考虑到节点针对交易验证的执行方式，该平面尚未充分利用全部网络带宽。需要说明的是，该问题可通过 BIP 152（致密区块中继）予以解决。

第二层称作共识平面，负责挖掘和实现共识。该层所面临的瓶颈主要是工作量证明算法中的种种限制。其中，鉴于分支（分叉）数量的增长，增加共识速度和带宽将会导致安全问题。

第三层称作存储平面，并负责存储账本。该层的主要问题是各个节点须持有一份账本的副本，进而导致效率下降，例如不断增长的带宽和存储需求。比特币则采用了修剪方法，节点运行时无须下载全部区块链。从存储角度来看，这可视为一项重大的改进。

列表中随后的内容是视图平面，并提出了一种优化方案，也就是说，比特币矿工不需要操作完整的区块链，可根据完整的账本构建视图，并作为系统的全部状态表现结果。这对于矿工操作而言已然足够。视图的实现将消除存储全部区块链的挖掘节点。

最后，Miller 还提出了侧平面，体现了链外交易这一概念；而支付或交易通道这一类概念则用于卸载参与者之间的交易处理，但主比特币区块链仍对此进行备份。

上述模型针对当前区块链设计采用结构化方式描述了相关问题及其改进措施。除此之外，近几年来还涌现出多种通用策略，例如以太坊和比特币，后续章节还将对此进行详细讨论。

12.1.1 增加区块链尺寸

对于提升区块链的性能（交易处理的吞吐量），该话题尚存在争议。当前，比特币每秒仅可处理 3~7 项交易，这也是处理微型交易时使用比特币区块链的主要抑制因素。此处，比特币区块尺寸硬编码为 1MB，当增加区块尺寸时，将承载更多的交易，从而缩短确认时间。针对区块尺寸的增加，存在多种比特币改进协议（BIP），包括 BIP 100、BIP 101、BIP 102、BIP 103 以及 BIP 109。在以太坊中，区块尺寸并不仅限于硬编码方式；相反，可通过 gas 限定条件予以控制。从理论上讲，以太坊中的区块尺寸并无严格限制（取决于 gas 量，并随时间增长）。另外，如果当前限制条件到达了前一区块的标准，还可针对后续区块适当提升 gas 限定量。

12.1.2 减少区块间隔时间

还有一项提议则是减少区块间的生成时间。也就是说，可降低区块之间的时间，进而提升区块的生成速度，但这会增加分支数量进而导致安全性降低。相应地，以太坊的区块生成时间大约为 14 秒并可适当增加。这可视作比特币区块链中的重大改进措施，也就是说，生成一个新区块将占用 10 分钟。在以太坊中，区块间较小的时间间隔常会导致孤立区块，这一问题可通过 GHOST 协议予以解决。但确定有效区块链时，仍会包含孤立

区块（uncles）。一旦以太坊改为权益证明，此类问题将不复存在——挖掘操作将不再必需，同时可即刻生成交易结果。

12.1.3 可逆的 Bloom 查找表

该方法旨在减少在比特币节点之间传输所需的数据量。可逆 Bloom 查找表（IBLT）最初是由 Gavin Andresen 提出的，而这种方法的主要吸引力在于不会产生比特币的硬分叉。其核心思想可描述为：无须节点之间传输全部交易，仅需传输同步节点交易池中不存在的交易。这将实现节点间快速的交易池同步操作，从而提升比特币网络的整体可扩展性和速度。

12.1.4 分片技术

分片并不是一种新的技术，已经被用于分布式数据库中，例如 MongoDB 和 MySQL。分片技术背后的关键思想是将任务分成多个块，然后由多个节点处理。这将提高吞吐量并降低存储需求。区块链中也采用了类似的方案，将网络的状态划分为多个分片。其中，状态通常包括余额、代码、nonce 和存储内容。分片是指运行于同一网络上的、区块链的松散耦合分区。关于分片间的通信，以及各分片历史上的共识，依然存在某些挑战性问题。目前，这也是一个开放的研究领域。

12.1.5 状态通道

状态通道是区块链网络上加速交易的另一种方法，基本思想是使用侧通道进行状态更新，并处理主链之外的交易。一旦状态完成，即被写回至主链中，进而从主区块链中卸载较为耗时的操作。状态通道的工作方式包含下列 3 个步骤：

- （1）部分区块链状态锁定于某个智能合约中，以保证参与者之间的协议和业务逻辑。
- （2）当前，在更新了状态的参与者之间，启动区块链之外的交易处理和交互过程。在此步骤中，几乎可以在不需要区块链的情况下执行任意数量的交易，因而可视为提升流程处理速度的主要原因，也是解决区块链扩展性问题的最佳候选方案。可能有人认为，这并非是一种真正的区块链解决方案，例如分片技术，但最终结果则是一个更快、更健壮的轻量级网络，这在微支付网络、物联网和许多其他应用程序中都非常有用。

- （3）一旦达到最终状态，状态通道随即关闭，最终状态将被写回至主区块链。在这个阶段，区块链的锁定部分也随之被解锁。

该技术用于比特币轻量级网络以及以太坊的 Raiden 中。

12.1.6 私有区块链

私有区块链本质上具有快速特征，因为不涉及真正的去中心化机制，同时网络上的参与者不需要执行挖掘操作；相反，私有区块链只能验证交易。这可以被看作是公共区块链中的可扩展性问题的解决方案。然而，这并不是解决可扩展性问题的最终方法。此外，应该指出的是，私人区块链只适用于特定领域和设置环境。

12.1.7 权益证明

与工作量证明算法相比。基于区块链的权益证明算法具有更快的执行速度。

12.1.8 侧链

侧链可间接地提升可扩展性，即允许多个侧链与主区块链协同运行，并可采用安全性稍差但速度更快的侧链执行交易，且仍与主区块链进行关联。这里，双向楔入则是侧链的核心概念，并支持父链至侧链之间的货币转移，反之亦然。

12.1.9 子链

子链是 Peter R. Rizun 提出的一种新技术，该技术基于各层中创建的弱区块（weak block），直至最终发现强区块（strong block）。其中，弱区块的定义方式可描述为：满足标准网络难度标准且无法挖掘的区块，但已完成了足够的工作量，并满足另一个较低难度的目标。矿工可以将弱区块分层叠加在一起建造子链，除非找到一个符合标准难度目标的区块。此时，子链关闭并成为强区块。该方法的优点在于：缩短了交易首次验证的等待时间，同时也减小了孤立块出现的机会，从而加速了交易处理过程。这也是解决可扩展性问题的一种间接方式，子链不需要任何软分叉或硬分叉来实现，但需要得到社区的认可。

12.1.10 树形链

在提升比特币可扩展性方面，其他建议还包括树形链，并将区块链布局从线性顺序模型调整为树形结构。该树形结构基本上是一棵源自主比特币链的二叉树。该方案与侧链类似，且无须修改主协议或增加区块尺寸，并显著地改善了交易的吞吐量。在该方案

中，区块链以分片形式呈现，并分布于网络中，进而实现可扩展性。除此之外，在树形链上，不再需要执行挖掘工作以对区块进行验证。相反，用户可独立验证区块头。然而，这一理念并不适用于实际产品，且进一步的研究工作仍在持续进行中。

除了上述讨论的一些通用技术之外，在其出版的 *On the Scalability and Security of Bitcoin* 一书中还提到了一些与比特币相关的改进措施。由于当前信息传播机制导致分叉出现，因而其实现源自如何进一步提升传播速度。相关技术涉及验证过程最小化、区块传播管线机制以及提升连通性。这一类变化无须对协议进行修改；相反，此类变化可在比特币节点软件中独立实现。关于验证最小化问题，需要注意的是，区块验证过程是导致传播延迟的原因。其背后的原因在于，节点一般会占用较长的时间在当前区块内验证区块和交易的唯一性。此处建议，一旦初始工作量证明和区块验证检测完毕后，节点即可发送存储消息。通过这种方式，仅需执行首个难度检测，且无须等待交易验证完成，进而改进传播过程。除了上述建议之外，区块传播管线机制也应引起足够的重视，该机制基于区块的可用性预测。其中，区块的可用性可事先声明，且无须等待生成实际区块，进而可减少节点间的往返时间。最后，交易发起者与节点间过长的距离也会降低区块的传播速度。经 Christian Decker 研究后表明，增加连通性可降低区块和交易的传播延迟。如果某一时刻比特币节点连接至多个节点，即可缩短节点间的距离，进而提高网络上的信息传播速度。

对于可扩展性问题，上述方法的部分或全部组合，可能会产生一种更为优雅解决方案。为了解决区块链的可扩展性和安全性问题，多项措施已经付诸于实践。例如，比特币隔离见证机制即可大规模提高扩展性，其中只需要使用一个软分叉即可。其背后的关键思想是 segwit，并将签名数据与交易分开，从而解决交易的扩展问题，同时可增加区块的尺寸。

另一项提议近期也引起了人们的关注，即基于微区块和领导者选举机制的比特币 NG，核心思想是将区块分成两种类型，即领导者区块和微区块。领导者区块负责工作量证明，而微区块包含了实际交易。相应地，微区块不需要任何工作量证明，并且由当选的领导者在各区块生成周期内生成。其中，区块生成周期由领导者区块发起，其唯一要求是利用所选领导者的私钥对微区块进行签名。微区块可经由所选领导者（矿工）高速生成，从而提高了性能和交易速度。

近期，Vitalik Buterin 在于上海召开的 Ethereum Devcon 2 会议上发表了一篇论文，并阐述了具有扩展特征的以太坊版本问题。该论文源自分片技术和权益证明算法实现的组合结果。此外，论文还制定了一些实现目标，包括基于权益证明的效率问题、快速的区块生成时间、经济目标、可扩展性、分片间的通信以及防审查机制。

12.2 隐 私 性

交易隐私是区块链的一个较为明显的特征，尤其是在公共区块链中，一切均处于透明状态。因此，此类属性也限制了区块链在不同行业的应用，在这些行业中，隐私往往显得十分重要，例如金融、医疗健康事业等。对于隐私问题，目前也涌现出了各种方案，某些方案已取得了较大的进展，例如不可区分性混淆（Indistinguishability Obfuscation, IO）技术、同态加密、零知识证明以及环签名。

12.2.1 不可区分性混淆技术

对于区块链中的隐私问题，此类加密技术可能会成为一种有效的解决方案，但该技术尚不适用于产品部署环节。不可区分性混淆（IO）支持代码的混淆操作，这也是密码学中一个非常成熟的研究课题。当应用于区块链中时，可作为一种无法破解的混淆机制，并将智能合约变成一个黑盒子。IO 背后的关键思想是研究人员提出的多重线性拼图方案，基本上是通过随机元素混合程序代码。如果程序按照期望方式运行，将生成预期的输出结果。Sahai 在其学术论文 *Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits* 中首次提出了这一理论。

12.2.2 同态加密

这种类型的加密技术可在加密数据上执行操作。想象一下，可将数据发送至云服务器进行处理，服务器处理后返回输出结果，且对所处理的数据一无所知。同态加密技术也是一个较为成熟的研究领域；但全同态加密（支持加密数据上的全部操作）尚不支持产品部署，但这一领域已经取得了重大进展。当在区块链采用该技术后，即可对密码文本进行处理，以支持交易的内部隐私性和机密性。例如，存储在区块链上的数据可以使用同态加密，无须解密即可对该数据进行计算，从而为区块链提供隐私服务。MIT Media Lab 在其 Enigma 项目中首次实现了这一概念。Enigma 是一个 P2P 网络，支持多方参与，并可在加密数据上执行计算，且无须显示与数据相关的任何内容。

12.2.3 零知识证明

如前所述，Zcash 中已经成功地实现了零知识证明。具体而言，为了确保区块链的隐

私，目前已经实现了 SNARK。该理念也可在以太坊和其他区块链中实现。Zcash 与以太坊之间的集成已经是一个非常活跃的研究项目，并由以太坊研发团队和 Zcash 公司运营。

12.2.4 状态通道

由于全部交易运行于区块链之外，且主区块链除了最终的状态输出结果之外无法看到当前交易，因而可采用基于状态通道的隐私，从而确保隐私和机密性的有效实施。

12.2.5 安全的多方计算

安全的多方计算并非新鲜事物，且基于以下观点：在秘密共享机制下，数据被分成多个分区，随后针对数据进行实际处理，而不需要在单台机器上重新构造数据，处理后产生的输出也在多方之间共享。

12.2.6 通过硬件提供保密性

可信赖的计算平台可提供某种机制，进而可在区块链上实现交易的机密性，例如 Intel 软件保护扩展（SGX），这允许代码在硬件保护环境运行，即受保护的执行区域。一旦代码在该区域中运行，即可生成一个 Intel 的云服务器所确认的引用证明。然而，受信的 Intel 将会导致某种程度的集中，并与区块链技术的真正精神背道而驰。尽管如此，该解决方案仍包含自身的优点，而且在实际操作过程中，许多平台已经开始使用了英特尔芯片，因此在某些情况下，Intel 方案依然是可以接受的。

若将该技术应用于智能合约，那么一旦节点执行了智能合约，即可生成引用并作为正确的证明以及有效的执行结果，而其他节点只需对其进行验证即可。通过可信执行环境（TEE），该思想可得到进一步扩展，并可以提供与保护区域（enclave）相同的功能，甚至可在具有近场通信（NFC）和安全元素的移动设备上使用。

12.2.7 Coinjoin

Coinjoin 技术通过对比特币交易采用交互式混合实现其匿名化操作，该思想从多个实体中构建单一交易，且不会改变输入和输出结果。另外，Coinjoin 移除了发送者和接收者之间的直接链接，也就是说单个地址不再与交易关联，这可能需要使用到用户身份的识别。Coinjoin 建立在多方合作的基础上，各方通过混合支付生成单一交易。因此，需要注意的是，如果 Coinjoin 方案中的任意参与者均未遵守构建单项交易合作所制定的承诺，

即未按照相关要求签署相关交易，将会导致拒绝服务攻击。在该协议中，无须设置独立的第三方受信机构，这不同于混合服务，即比特币用户之间的受信第三方机构或中介机构，且支持交易的混编功能。其中，交易的混编可防止支付行为跟踪或链接至某个特定的用户。

12.2.8 机密交易

机密交易利用 Pedersen 承诺提供保密性。承诺方案负责对某些数值提供相关保证，并对其进行加密，同时还可在后续操作过程中对其予以显示。当设计承诺方案时，须满足两个属性，即关联属性和隐藏属性。其中，关联属性确保承诺方之后无法改变所选值；而隐藏属性则使对方无法获取承诺方提供的原始值。Pedersen 支持累加操作，并具有“交换律”这一类特性，因而在比特币交易的机密性方面特别有用。换句话说，它支持数值的同态加密，并允许在比特币交易中隐藏支付值，这一概念已在 Elements 项目中有所实现（对应网址为 <https://elementsproject.org/>）。

12.2.9 MimbleWimble

自 MimbleWimble 方案在比特币的 IRC 频道出现后，即获得了大量的人气。MimbleWimble 扩展了机密交易和 Coinjoin 这两个概念，支持交易聚合且无须任何交互行为。然而，MimbleWimble 不支持比特币脚本，以及标准比特币协议中的各种特性，因而与现有的比特币协议并不兼容。因此，MimbleWimble 仅实现于侧链比特币，或自身中的替代加密货币。

MimbleWimble 方案一次性地解决了隐私和可扩展问题，采用 MimbleWimble 技术生成的区块并不包含交易内容，这与传统的比特币区块链截然不同。相反，这一类区块由 3 个列表组成，即输入列表、输出列表以及由签名和输入、输出差异等构成的列表。其中，输入列表基本上表示为之前输出内容的引用；输出列表则包含了加密交易输出结果。这一类区块均可通过签名、输入以及输出结果经由节点进行验证，进而保证区块的合法性。与比特币相比，MimbleWimble 交易输出结果仅包含公钥，而新、老输出结果之间的差异则由交易的全部参与者进行签名。

12.3 安 全 性

即使区块链具备了一定的安全性，并在区块链网络上采用了非对称或对称加密，但

某些时候依然会存在一些缺陷导致区块链的安全性降低。

经研究人员证实，相关示例包括交易的可扩展问题、Eclipse 攻击以及比特币中的双重支出问题。其中，交易的扩展问题可能导致重复取款或存款，即攻击者在网络确认之前修改交易的唯一 ID，从而造成交易并未发生的假象。对此，可结合使用 BIP 62 和 segwit 解决此类问题。需要注意的是，此类问题仅出现于未确认交易中。也就是说，各项操作处理均在未确认交易上进行。相应地，常见的应用程序均在确认后的交易中进行，因而不会产生任何问题。

另外，比特币中的信息 eclipse 攻击常会导致双重支付问题，即欺骗比特币节点，使其仅与攻击者节点 IP 进行连接，从而导致其受到“51%攻击”。目前，这一问题已在比特币客户端 v0.10.1 版本中得到了一定程度上的解决。

12.3.1 智能合约安全性

最近，研究人员在智能合约安全领域展开了大量的工作，特别是智能合约的形式验证问题，这一切均由 DAO 攻击造成。此处，形式验证表示计算机程序的验证过程，并确保满足特定的形式语句。目前，形式验证问题还是一个较新的概念，针对其他语言，存在多种工具可完成这项任务。例如，Frama-C 可用于分析 C 语言。形式验证背后的核心理念是将源程序转换为一组验证方可理解的语句。对此，Why3 是一类较为常见的工具，Solidity 语言中的形式验证器也使用了该工具。另外，Solidity 中已经配置了测试版本的验证器。目前，智能合约的安全性问题受到了极大的重视，同时还采纳了其他多项措施，并设计相关方法分析 Solidity 程序和 bug。Oyente 是研究人员近期（至成书时止）发布的一款创新工具，并在发表的论文 *Making Smart Contracts Smarter* 中提到了这一工具。除此之外，论文还对智能合约中的安全问题予以揭示和分析。相关问题包括交易排序依赖性、时间戳依赖性、错误处理异常（例如堆栈调用深度的限制条件）以及重入漏洞。其中，交易排序依赖性问题基本上指，合约感知状态与执行后的变更状态不一致。这一缺陷体现了一种竞争条件，也称作前置（frontloading），其原因在于：区块中的交易顺序可被操控。鉴于全部交易首先出现于内存池中，因而交易在纳入区块前可被监视。这也使得某项交易可在另一项交易之前被提交，进而控制智能合约的行为。

时间戳依赖性问题是指，在某些场合下，区块的时间戳可用作合约中制定的决策源，但时间戳依然可被矿工予以操控。调用栈深度限制条件则是另一个问题，且源自下列事实：EVM 的最大调用栈深度为 1024 帧。如果在合约执行时达到了栈深度，则在某些场合下，发送或调用指令将会失效，从而导致资金无法被支付。对此，EIP50 硬分叉（分支）

中解决了调用栈深度问题。另外，重入 bug 则被 DAO 攻击所利用，导致数百万美元被转置子 DAO 中。基本上讲，重入 bug 意味着，某个函数在其上一次（首次）调用之前被反复调用。在 Solidity 智能合约的 Ether 提款功能中，这显得尤为不安全。

除了上面提到的 bug 之外，在编写合约时还需要注意一些其他问题，其中包括：如果将资金发送到另一个合约，须对此谨慎处理——发送过程可能会失败，即使采用了全方位的防范机制，也不会起到任何作用。

其他软件 bug 也应引起足够的重视，如整数上溢和下溢问题。整数变量应在 Solidity 实现中予以谨慎处理。例如，在一个简单的程序中，如果采用 uint8 解析包含超过 255 个元素的数组元素，将会产生无限循环。这是因为 uint8 被限定为 256 个数字。

下面将分别使用 Why3 和 Oyente 展示两个合约验证的例子。

12.3.2 Why3 形式验证

对于 Solidity 代码的形式验证，Solidity 浏览器已将其设置为特性。首先，代码转换为验证器可以理解的 Why3 语言。下面的示例中显示了一段简单的 Solidity 代码，定义了变量 z 作为 uint 的最大值。代码运行后将返回 0 值，因为 uint z 将会溢出并从 0 重新开始。图 12.1 显示了 Why3 的验证结果。

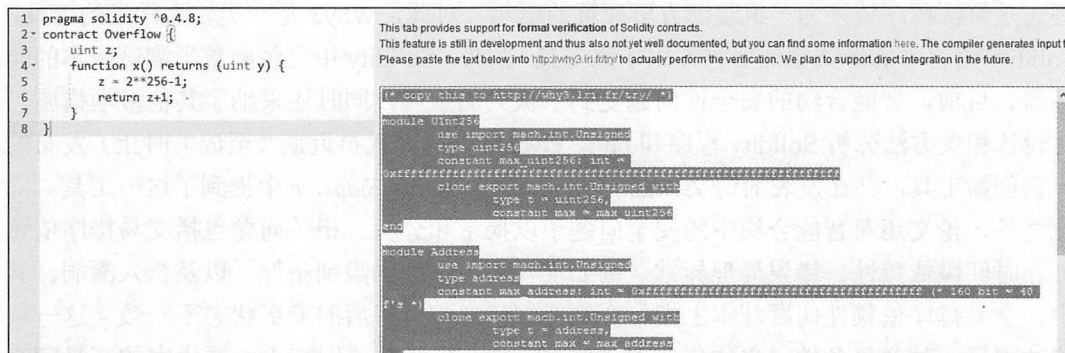


图 12.1 包含 Why3 形式验证的 Solidity 在线编译器

一旦 Solidity 被编译并在形式验证选项卡中可用，即可被复制到 Why3 的在线 IDE 中，对应网址为 <http://why3.lri.fr/try/>。下面的示例成功地检查并报告了整数溢出错误。需要注意的是，Why3 工具仍处于开发中，但依然可发挥其应有的功效。另外，该工具或者其他类似工具并非万能，即使形式验证也是如此，因为首先应该定义相应的规范，如图 12.2 所示。

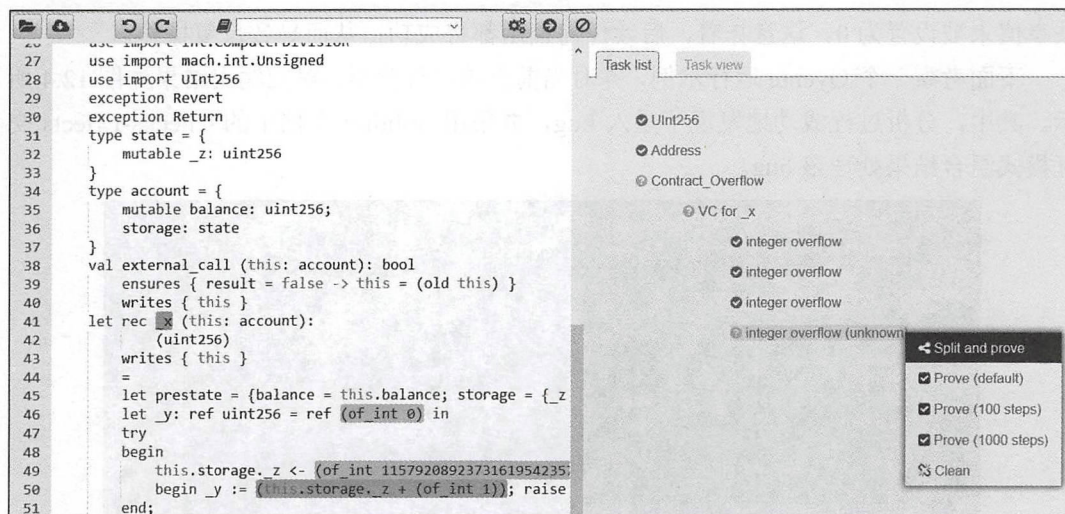


图 12.2 Why3 工具

12.3.3 Oyente 工具

目前，作为 Docker 镜像，Oyente 可实现快速的测试和安装过程。读者可访问 <https://github.com/ethereum/oyente> 下载并测试 Oyente。在下面的示例中，将测试一份 Solidity 文档中的简单合约，其中包含重入 bug。相应地，Oyente 成功地分析了代码并发现了其中的错误，如图 12.3 所示。

```

1 pragma solidity ^0.4.0;
2 contract Fund {
3     mapping(address => uint) shares;
4     function withdraw() {
5         if (msg.sender.send(shares[msg.sender]))
6             shares[msg.sender] = 0;
7     }
8 }

```

图 12.3 源自 Solidity 文档中的合约，其中包含了重入 bug

上述示例代码包含了一个可重入的 bug，也就是说，如果某个合约与另一个合约或 Ether 传输进行交互，那么实际上则是将控制权移交给另一个合约。这也使得被调用的合约回调该合约的函数，且无须等待完成。例如，该 bug 可反复调用前一个示例中显示的 `withdraw` 函数，从而多次获取 Ether。这种情况是有可能发生的，因为在函数结束之前，

共享值未被设置为 0。这意味着，后续任何调用都将成功，从而导致反复操作。

下面考察一个 Oyente 运行示例，并对当前合约进行分析，对应输出结果如图 12.4 所示。其中，分析过程成功地发现了重入 bug，并采用 Solidity 文档中的 Checks-Effects 交互模式组合结果处理该 bug。

```

root@fa9ef6ac8455: /home/oyente/oyente
(venv)root@fa9ef6ac8455:/home/oyente/oyente# python oyente.py a1.sol
Contract Fund:
Running, please wait...
===== Results =====
CallStack Attack:      False
THIS IS A CALLLLLLLLLLL
{'path_condition': '[Iv >= 0, init_Is >= Iv, init_Ia >= 0, If(Id_0/
26959946667150639794667015087019630673637144422540572481103610249216 ==
1020253707,
1,
0) !=
0, Not(Iv != 0)], 'Is': Is, 'Iv': Iv, 'some_var_1': some_var_1, 'Id_0': Id
_0, 'Ia_store_some_var_1': Ia_store_some_var_1, 'Ia': Ia]}

This is the global state
{'Ia': {'some_var_1': 0}, 'miu_i': 3L, 'balance': {'Ia': init_Ia + Iv, 'Is
': init_Is - Iv}}
{64: 96, 0: Is & 1461501637330902918203684832716283019655932542975, 32: 0}

CALL params

Is & 1461501637330902918203684832716283019655932542975

Ia_store_some_var_1

=>>>>> New PC: []

Reentrancy_bug? True

Added True
Concurrency Bug:      False
Time Dependency:      False
Reentrancy bug exists: True
===== Analysis Completed =====
(venv)root@fa9ef6ac8455:/home/oyente/oyente#

```

图 12.4 Oyente 工具检测 Solidity 中的 bug

12.4 本章小结

本章介绍了区块链技术的安全性、保密性和隐私方面的内容。在不同行业中，隐私问题可视为公共区块链应用发展的一个主要抑制因素。接下来讨论了智能合约的安全问

题，这是一个目前非常热门的话题，同时涵盖了深刻而广泛的内容。本章在各个方面都进行了简要的介绍，并为该领域进一步的研究打下了坚实的基础。例如形式验证问题、相关示例以及可用工具。应该注意的是，相关工具仍处于开发中，尚缺乏令人满意的操作特性。同样，开发文档也有待进一步完善。因此，我们鼓励读者密切关注形势的发展，特别是与以太坊相关的形式验证和开发内容，因为这一领域发展十分迅猛。区块链安全领域，尤其是智能合约安全领域，目前已发展得比较成熟，这一主题甚至可独立成书。学术界和商业领域中的许多专家和研究人員也在密切关注着这一领域，相信很快就会出现许多自动化工具对智能合约进行验证。

第 13 章 发展现状和未来趋势

区块链技术将改变我们对日常事务的处理方式，对现有的商业模式提出了挑战，并承诺在节约成本、提高效率和透明度方面获得巨大利益。本章将探讨区块链技术的最新发展、新趋势、问题以及未来的预测。此外，本章还将详细讨论一些与开源技术相关的问题及其改进措施。

13.1 新趋势

当前，学术界和商业界均对区块链技术产生了浓厚兴趣，区块链技术正处于快速变化和高速发展阶段。随着技术的不断成熟，业界也涌现出了一些新的趋势。例如，私有区块链最近因其在金融领域的特殊应用而获得了广泛的关注。此外，企业区块链则是另一种新趋势，旨在开发满足企业级效率、安全性和集成需求的区块链解决方案。

13.1.1 基于应用程序的区块链（ASBC）

目前，ASBC 引起了人们的广泛关注。其中，区块链或分布式账本针对某一特定应用而开发，且主要集中于某个特定领域，例如 `everledger`。作为区块链，`everledger` 用于提供不可变的跟踪历史以及审计跟踪。该方法可以防止任何欺诈行为，因为与所有权、真实性和价值相关的内容均经过验证并记录在区块链上。这对保险和执法机构来说具有很大的应用价值。

13.1.2 企业级区块链

考虑到隐私性和可扩展性，区块链的原始形式尚不适用于企业级应用。企业级区块链也是近期涌现的新鲜事物，一些公司开始关注这一领域，并在企业级别上尝试部署和集成方案。其中，测试、文档、集成以及安全问题均已得到妥善解决，并尝试以最小代价（仅作较少改动，或者不作任何变化）实现企业级应用。这与公共区块链形成鲜明对比，后者是不受监管的，并且不符合特定的企业级安全需求。这也意味着，企业级区块链通常是在私有配置中实现的；尽管如此，公共企业级区块链的实现也不失为一种选择。

方案。2016 年，许多科技初创企业开始提供企业级的区块链解决方案，如 bloq、tylmez、chain 等。这一趋势仍在持续增长中，2017 年将会出现更多这样的技术。

13.1.3 私有区块链

考虑到隐私性和机密性，须开发相应的分布式私有账本，以供一组受信参与者使用。作为公有区块链，鉴于其开放特性以及缺乏应有的安全性，因而并不适用于诸如金融、医疗以及法律等相关行业；相比之下，私有链则解决了此类问题，用户可从中体验到区块链的各种优点，同时满足安全性和隐私性要求。由于公有区块链未提供隐式和机密性服务，因而缺少应有的安全特征；而私有区块链则允许参与者或参与者子集完全控制系统，因而适用于对隐私和控制要求较高的行业，例如金融行业。

以太坊可用于私有模式和公有模式，另外，诸如 Hyperledger 和 Corda 等项目已作为私有链被开发。

13.1.4 初创公司

近年来，许多科技初创企业都发布了区块链项目，并提供了针对该技术的解决方案。特别是 2016 年，提供区块链咨询和解决方案的初创企业数量与日俱增。

13.1.5 浓厚的研究兴趣

区块链技术在学术界和商业领域都引起了人们浓厚的研究兴趣。近年来，人们的关注度普遍增加，世界各地的主要机构和研究人员均对此展开了研究工作。这也说明，区块链技术可以帮助企业提高效率，降低成本并提升事务的透明度。相比之下，学术界则是围绕着密码学、共识机制、性能以及区块链中其他限制问题解决难题。例如，UCL 下属部门“UCL 区块链技术研究中心”即专注于区块链方向的技术研究。另一个例子是 ETH Zurich 分布式计算小组，同时发表了多项与区块链技术相关的研究成果。近期，一家名为 Ledger Journal 的期刊发表了首篇学术论文，读者可访问 <http://www.ledgerjournal.org/ojs/index.php/ledger> 以了解更多内容。目前，在各种学术和商业机构中，均设置了区块链研究和开发的相关团队和部门，这也从侧面反映出了区块链技术的热度。预计 2017 年以后，这一领域还会呈现更多的研究和发展成果。另一家名为 IC3（加密货币与合约）的组织机构也致力于智能合约和区块链技术的研究工作。IC3 的目标是解决区块链和智能合约的性能、机密性和安全问题，并通过多个项目来解决这些问题。对此，读者可访问 <http://www.initc3.org/> 以了解 IC3 的项目信息。

13.1.6 标准化

目前，区块链技术尚缺乏足够的成熟度，尚无法与现有系统实现完美集成。即使利用现有技术，两个区块链网络之间也无法实现流畅的通信。对此，标准化有助于提高区块链技术的互操作性、可移植性以及可集成性。近期较为引人关注的事件是设立了 ISO/TC 307 技术委员会，其范围包括标准化的区块链和分布式账本技术。该委员会的目标是在用户、应用程序和系统之间提升互操作性和数据交换能力。另一方面，最近创办的联盟和开放源码协作机构，例如 R3 和 Hyperledger，通过分享理念、工具和代码，实现了区块链技术的标准化。R3 联手 80 多家银行组成财团，且彼此间具有类似的目标，这在某种程度上也可视为一种标准化结果。另一方面，Hyperledger 设置了一个参考架构，可以用来构建区块链系统，并得到 Linux 基金会和其他来自该行业的参与者的支持。另一个例子是链开放标准，这是为金融网络开发的协议，并发布了 OS1 标准。该标准是在世界上主要金融机构合作的基础上形成的。该标准支持快速的交易结算以及即时 P2P 交易路由，旨在满足区块链技术中的监管、安全和隐私要求。另外，OS1 还为智能合约开发提供了一个框架，使得参与者可轻松地满足 AML 和 KYC 的需求。

智能合约的标准化工作也源自 Lee 等人发表的一篇开创性的论文，其中正式定义了智能合约模板，并为智能合约的后期研发绘制了发展蓝图。读者可访问 <https://arxiv.org/abs/1608.00771v2> 阅读该论文。另外，第 6、12 章也对这一话题有所讨论。

上述各项工作均表明，在不久的将来，行业内的标准将会使区块链技术变得更加简单、快捷。同时，由于标准的实施将消除互操作性等障碍，因而区块链行业将会呈指数级增长。

13.1.7 改进措施

在过去的几年里，人们对现有区块链的进一步发展提出了各种改进和建议，相关内容大多针对安全漏洞，并消除了区块链技术中的固有局限性。区块链技术仍存在某些制约条件，如可扩展性、隐私和互操作性。在区块链衍变为主流技术之前，此类问题都需要予以解决。最近，在区块链技术中已经讨论了大量的可扩展问题，第 12 章也曾对此有所提及。此外，开发人员还会定期地针对某些问题提出改进意见，例如 BIP（比特币改进建议）和 EIP（以太坊改进建议），以解决这些系统中的各种问题。本章后续内容还将讨论一些近期值得关注的改进建议。另外，区块链技术在某些方面也取得了迅速发展，例如状态通道，相信很快会发展成一种更加成熟和实用的技术。

13.1.8 具体实现

近年来，特别是在 2016 年，基于区块链技术的各种概念证明已日趋完善，同时涌现出了一些特定的应用程序实现，如物联网和 everledger；但在其他领域，情况则不容乐观。目前，尚未实现真实的、端到端的应用，特别是在金融行业。鉴于大量的概念证明已较为完善，因而最终实现似乎并不会太遥远。下一阶段是在实际场景中实现这些方案。例如，最近，7 家银行已同意建立一个数字交易链（DTC），以简化贸易融资流程。

13.1.9 企业联合体

目前，越来越多的企业联合体、委员会和开源项目如雨后春笋般地出现。例如 R3，它与世界上最大的金融组织结成了联盟。

13.1.10 解决方法

针对区块链技术，由于社区的蓬勃发展，各种各样的挑战性问题也随之出现。例如状态通道，这一概念被用作对区块链的可扩展性和隐私问题的响应。当使用状态通道时，比特币的闪电网络和以太坊的 Raiden 基本已处于实现阶段。此外，各种区块链解决方案也层出不穷，如 Kadena，该方案直接解决了区块链的机密性问题。其他一些概念，如 Zcash、Coinjoin 和机密交易也处于成熟阶段，前述章节曾对此有所讨论。这一趋势还将在未来几年持续加深，即使相关挑战问题在区块链技术中均已得到解决，改进和优化的步伐也永远不会停止。

13.1.11 技术融合

其他技术与区块链的融合也带来了巨大的收益。在核心内容上，区块链提供了灵活性、安全性和透明度，当与其他技术结合在一起时，会产生一种非常强大的功效，且彼此间相互补充。例如，物联网与区块链（如完整性、去中心化和可扩展性）结合时具有明显的优势。人工智能也有望从区块链技术中获益。事实上，在区块链技术中，人工智能可以通过自主智能体（AA）的形式实现。后续内容还将对技术融合方面的示例予以展示。

13.1.12 教育发展状况

尽管区块链技术已经引起了技术人员、开发人员和各行业科学家的极大兴趣，但目

前仍缺乏正规的学习资源和教材。针对这一项新技术，许多著名的教育机构，如普林斯顿大学即开授了相关课程。具体而言，普林斯顿大学通过在线教育方式开设了加密货币和数字货币课程，受到了学生的普遍欢迎。另外，许多私人教育机构也提供了类似的在线课堂培训课程。随着区块链技术的普及和被接受程度不断增强，相信在不久的将来会看到更多类似的成果。

13.1.13 就业前景

最近，招聘市场出现了一种趋势，即企业正在大量寻找区块链设计专家和开发人员，这与金融行业尤其相关。近期，许多初创企业和大型组织也开始雇佣区块链专家。当然，随着技术的不断成熟、普及，这一趋势将会处于持续增强的态势。随着技术的进步，越来越多的开发人员通过自学方式获取了宝贵的经验，或者参加培训机构开设的相关课程。

13.1.14 密码经济学

新的研究领域也伴随着区块链出现，其中最著名的是密码经济学，并致力于去中心化数字经济协议方面的研究。随着区块链和加密货币的出现，这一领域的研究也呈增长之势。密码经济学由 Vitalik Buterin 首次提出，其定义涉及数学、密码学、经济学和博弈论等综合内容。

13.1.15 密码学研究

尽管在比特币发明之前，密码学一直是人们关注和研究的领域，但区块链技术再次引发了人们对这一领域的关注。随着区块链和相关技术的出现，密码学的受关注度显著增加。特别是在金融密码学领域，新的研究成果也层出不穷。对于区块链应用，研究领域涉及零知识证明、全同态加密和功能加密等技术。Zcash 在应用级别上首次实现了零知识证明。可以看到，区块链和加密货币有助于推动密码学的发展，特别是金融密码学。

13.1.16 新的编程语言

此外，开发智能合约的编程语言也开始受到人们的关注，主要涉及特定领域中的编程语言。例如，以太坊的 Solidity 语言和 Kadena 的 Pact 语言。这仅仅是一个开始，相信随着技术的进步和发展，还将涌现出其他新型语言。

13.1.17 硬件研究和开发

2010 年，针对比特币开采，当人们意识到目前的方法较为低效时，矿工开始转向优化采矿硬件，包括使用图形处理单元（GPU），在 GPU 达到极限后转向现场可编程门阵列（FPGA）。在此之后，专用集成电路（ASIC）的出现，大大增加了采矿能力。这一趋势预计将进一步发展，目前一些研究成果可进一步优化 ASIC，例如并行化和减小晶粒尺寸。此外，预计 GPU 编程也会呈增长趋势。新的加密货币大多数采用了工作量证明算法，这些算法可以从 GPU 处理能力中获益。例如，最近 Zcash 引发了人们对 GPU 采矿设备的兴趣，以及使用 NVidia CUDA 和 OpenCL 的程序设计。其目的是为了优化采矿作业，同时使用多个 GPU。此外，在可信计算硬件方面也涌现出了一些研究成果，如英特尔的软件保护扩展（SGX），并以此解决区块链的安全问题。此外，英特尔的 SGX 已经被用于一种新的共识算法，称为消逝时间量证明（PoET），前述章节曾对此有所讨论。另一个项目是“21 比特币计算机”，作为开发人员学习比特币技术的平台，以使读者针对比特币平台开发相应的应用程序。

硬件方面的发展和研究处于持续升温状态，相信不久的将来会出现更多的解决方案以供人们借鉴。

13.1.18 形式方法和以及安全研究

对于安全问题和智能合约编程语言中的漏洞，在生产部署之前对智能合约进行形式验证和测试变得十分必要，其中包括针对以太坊 Solidity 语言的 Why3；另一个例子则是针对智能合约保密性的 Hawk。

13.1.19 区块链的替代方案

近年来，随着区块链技术的进步，研究人员开始考虑创建其他平台，并可提供区块链中的保证机制和服务，但并不涉及区块链。针对于此，R3 发布了 Corda。实际上，Corda 并非真正的区块链——未采用包含交易的区块概念；相反，Corda 是基于状态对象这一概念，根据网络参与者的要求和规则，在 Corda 网络中进行传输，进而体现网络的最新状态。其他的例子还包括 IOTA，作为 IoT 区块链，它使用一个有向无环图形（DAG）作为一个分布式的账本（名为 Tangle），而不是传统的包含区块的区块链。该账本声称已经解决了可扩展性问题，以及更高级别的安全问题，甚至可以防止基于量子计算的攻击。应该指出的是，比特币在某种程度上也受到了保护，因为量子攻击只能在外露的公钥上

工作，只有在发送和接收交易时才会显示在区块链上。如果公钥未被显示，在未使用的地址或仅用于接收比特币的地址中，则可以保证量子安全。换句话说，为每项交易使用不同的地址可有效地防止量子攻击。同样，在比特币中，如果必要，可方便地更改为另一个量子签名协议。

13.1.20 互操作性

关于区块链的互操作性，近期实现了跨区块链系统。例如 Qtum，它是一个与以太坊和比特币区块链兼容的区块链。针对价值转移，Qtum 采用了比特币的 UTXO 机制；而对于智能合约，则使用了以太坊虚拟机。这意味着在不需要任何改变的情况下，以太坊项目可以移植到 Qtum 上。

13.1.21 区块链服务

随着云平台的成熟，许多公司已经开始提供区块链服务（BaaS）。最突出的例子是微软的 Azure，其中以太坊区块链作为服务予以提供；另外，IBM 的 Bluemix 平台则用于提供 IBM BaaS。这一趋势预计将在未来几年内持续加强，更多的公司将提供 BaaS。EgaaS（电子政府服务）则是另一个例子，实际上仍可视作 BaaS，仅是针对治理功能提供的、基于特定应用的区块链。该项目旨在组织和控制零文档、零开销的相关活动。

13.1.22 减少耗电量

从比特币的区块链中可以明显看出工作量证明机制的效率低下。当然，该计算保证了比特币网络的安全，同时也浪费了大量电能。为了减少这种浪费，可选择诸如权益证明算法这一类绿色方案，此类方案通常优于比特币的工作量证明算法。这一趋势预计会持续增长，尤其是以太坊权益证明算法。

13.2 改进协议

比特币和以太坊这两种主要的区块链技术均包含了相关机制，并可对现有协议进行改进，即改进协议（BIP 和 EIP）。这两种机制都允许来自世界各地的开发者和爱好者参与，并帮助比特币和以太坊逐渐发展成为一种更加成熟和安全的技術。后续内容还将对区块链的改进建议加以讨论。

13.2.1 BIP

本节主要讨论某些最新的 BIP。

1. BIP 152

作为改进建议，BIP 152 可在比特币网络中引入紧凑的区块，以节省带宽。在当前的状态下，比特币协议的带宽效率并不高。针对这一问题，该协议可在节点之间转发紧凑的区块。为了实现该协议，需对相关内容做适当调整，例如新的数据结构和消息。目前，该提案仍处于起草状态。读者可访问 <https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki> 以了解更多内容。

2. BIP 151

这一改进协议旨在引入 P2P 通信加密，进而改善比特币协议中的各种安全问题，例如流量分析和控制。该提案将有助于阻止通过源 IP 和比特币网络中的交易内容来识别用户。在 BIP 中，出现了一些新的请求消息类型，并可在参与节点中进行加密。读者可访问 <https://github.com/bitcoin/bips/blob/master/bip-0151.mediawiki> 以了解更多内容。

3. BIP 150

该协议提供了一种对等身份验证方法，可对中间人攻击进行处理，并确保对等点连接到合法的节点。读者可访问 <https://github.com/bitcoin/bips/blob/master/bip-0150.mediawiki> 以了解更多内容。

4. BIP 147

该协议为了解决虚拟堆栈元素的扩展性问题。签名扩展性是一个众所周知的问题，并可以通过合法节点予以解决，即简单地修改与交易关联的签名。BIP 147 是一种共识层级别的改进方案，需要采用软分叉予以实现。读者可访问 <https://github.com/bitcoin/bips/blob/master/bip-0147.mediawiki> 以了解更多内容。

5. BIP 146

BIP 146 旨在解决签名编码的扩展问题，并通过软分叉予以实现，这将解决与 ECDSA 签名编码机制中基本限制相关的签名扩展性问题，这种限制是 ECDSA 中所固有的。其次，如果签名失败，验证脚本评估仍将继续，并允许在签名上设置任何值。因此，BIP 146 主要用于解决此类问题。读者可访问 <https://github.com/bitcoin/bips/blob/master/bip-0146.mediawiki>

以了解更多内容。

除此之外，还存在其他 BIP 可用于改进比特币协议。本节仅介绍了 5 项最新协议。近年来，共计发布了 90 多项比特币改进协议，而且这一数字还在持续增长中。

13.2.2 EIP

以太坊包含自己的改进协议版本，称作 EIP。下面考察一些近期的 EIP。

1. EIP 170

该改进协议旨在解决某些漏洞问题，此类漏洞往往会导致 gas 的过量使用。在预处理阶段，或者代码的磁盘读取阶段，较大的合约代码尺寸往往会引发这一类问题。对此，协议将合约代码限定为 23999 字节。关于协议的改进，还涉及了某些更深层次的讨论，但目前尚未予以实现。

2. EIP 150

这一改进方案是为了响应以太坊区块链上的拒绝服务攻击。因此，自以太坊区块链的 2463000 区块起，区块链实现为硬分叉。该 EIP 试图解决由交易垃圾邮件攻击所引发的问题，例如，反复调用 EXTCODESIZE 操作码，从而导致区块链验证过程减缓。EIP 旨在增加操作码的 gas 消费，以阻止重复使用这些操作码而引发的拒绝服务攻击。

3. EIP 161

该协议将清除以太坊区块链上的空账户，以解决以太坊区块链中的拒绝服务攻击问题。其中，攻击者创建了大量的空账户。这可能是因为创建此类账户的成本较低。当前过程清空状态字典树，即通过账户的 CALL 指令移除空账户。

4. EIP 160

该建议方案将增加 EXP 操作码的开销，其背后的核心思想是使 EXP 操作开销与操作的复杂度成正比。该方案有助于阻止调用计算量较大的操作时引起的拒绝服务攻击。

5. EIP 155

针对主以太坊区块链上的不同链，该协议可消除重启交易。例如，如果以太坊网络上生成了某项交易，则可在以太坊经典网络上重启，而 EIP 155 则解决了这一问题。而且，任何已在测试网络上产生的交易都不能在主网络重启。

上述全部改进协议均会生成一个硬分叉，并实现为针对最新的以太坊拒绝服务攻击的一种响应。作为一种关键机制，改进协议对于以太坊区块链引入了最新的改进措施，

以及协议级别的问题修复。相应地，以太坊区块链中实现（或提出）了 11 种改进协议。读者可访问 <https://github.com/ethereum/EIPs>/获取 GitHub 上的全部文档。

13.3 其他挑战性问题

除了安全与隐私，第 12 章中也有所提及，在区块链成为主流技术之前，还存在一些问题需要予以解决，包括监管、政府控制、不成熟的技术、与现有系统的集成以及实施成本等问题。

监管问题被认为是需要应付的最大挑战之一，其核心问题是区块链以及不被任何政府视为法定货币的加密货币。尽管在某些情况下，加密货币已被归类为美国和德国货币，但它仍未被视作一种正常货币。此外，在目前环境下，区块链并不被认为是金融机构可以使用的平台。另外，金融监管机构还没有将其视为可以被授权使用的平台。然而，世界各地的监管当局已开始对此提出了各种各样的措施以及相关的规章制度。目前，比特币尚不受监管，尽管政府曾试图对比特币征税。在英国，根据欧盟增值税指令，比特币交易不受增值税（VAT）的限制，但在英国退出欧盟后，这种情况可能会发生变化。然而，资本增值税（CGT）在某些情况下仍然适用。

英国金融行为管理局（FCA）最近批准了一些使用区块链技术的公司，这也意味着，金融监管部门很快会对此类公司进行监管。

人们普遍担心区块链技术还没有真正做好产品部署的准备。虽然比特币区块链已经发展成为一个相对成熟的区块链平台并被用于生产环境，但该技术并不适合每一种场合，在诸如金融和健康等敏感的行业中尤其如此。然而，这一状况很快会得到改变，本章前述内容曾对此有所讨论，如可扩展性和隐私问题。安全性也是许多研究人员强调的另一个普遍关注的问题，这对于财政和卫生部门十分重要。欧盟网络和信息安全机构（ENISA）最近的一份报告特意强调了亟待解决的分布式账本问题。报告中提到的一些问题包括：智能合约管理、密钥管理、反洗钱和反欺诈工具。此外，报告中还强调了对监管、审计、控制和治理的各项要求。

与现有遗留系统的集成也是一个主要问题。目前，区块链与现有金融体系之间的整合方式尚不完善。

现有问题或多或少与管理、安全性和互操作性有关。另外，与现有系统的集成可以通过多种方式进行。

13.4 负面影响

区块链技术具备反审查、去中心化等属性，进而可增加透明度并提升效率，但这种技术在某种程度上是不受监管的，这也意味着可被罪犯用于非法活动。例如，考虑下列情形：如果在互联网上发布一些非法内容，则可立刻通知相关职能部门和网站服务提供商，进而消除不良影响——但这在区块链中是无法实现的。区块链上的任何内容几乎无法进行还原。这意味着，一旦在区块链上发布了任何不可接受的内容，此类内容均无法被删除、关闭。因此，这也是区块链技术面临的严重挑战，似乎在这种情况下，相应的监管和控制措施是有益的，但是处于监管状态下的区块链显然违背了该技术的初衷。或许，可先期制定监管法规，随后考察区块链技术是否可适应这种情况。这一类措施可能并不明智，同时可能会抑制区块链技术的创新和进步。对此，一种做法是让区块链技术在开始阶段自由发展，就像互联网一样，当达到临界点时，管理机构可对其实现和应用进行监管。

例如，Dark Web 与比特币一起使用以执行非法活动。又如，在互联网上销售非法毒品的 SilkRoad，则使用比特币支付，而使用“洋葱”（onion）网址 Dark Web 则只能用 Tor 来显示。尽管经过执法机构数月的努力，SilkRoad 已被关闭，但类似的网站不断地浮出水面。另外，某些替代方案也提供了类似的服务。总而言之，问题依然存在。假设 IPFS 和区块链上出现了一个非法网站，目前尚无简单的操作方法可关闭该网站。很明显，缺乏应有的控制和监管导致犯罪活动盛行，像 SilkRoad 这样的问题也会继续出现。像 Zcash 这样的完全匿名交易功能可以为罪犯提供另一层保护，但该技术可能在其他合法场合中非常有用。因此，这取决于这项技术的具体使用者。匿名性在很多情况下均具有明显的优势，例如医疗行业，病人的记录应该受到保护并呈现为匿名状态；但如果罪犯也以此隐藏其行为，那么匿名性亦不再合理。

一种解决方案是引入智能体或 AA，甚至是嵌入其中的、监管逻辑程序编写的合约，通常是由监管机构和执法机构编写的，并以区块链作为一种治理和控制的手段。例如，区块链可以采用下列方式进行设计：每一份智能合约都必须通过一个控制器合约审查代码逻辑，并提供一种管理机制控制智能合约的行为。另外，也可让每份智能合约代码接受监管部门的检查，一旦智能合约代码有一定程度的可靠性，并以监管机构颁发的证书形式出现，即可部署在区块链网络上。这一类二进制签名概念类似于代码签名，即通过数字签名作为一种手段确认代码具有真实性，而非恶意代码。这种想法更适用于半私人或受监管的区块链，例如金融领域。其中，监管当局需要施加一定程度的控制行为。这

也意味着，在第三方（监管机构）中需要建立一定程度的信任机制，但这偏离了完全去中心化这一概念。为了解决这一问题，区块链自身可以提供一個去中心化的、透明的、安全的证书颁发和数字签名机制。

13.5 区块链研究

虽然近年来在区块链技术上出现了一些重大的创新成果，但该领域的进一步研究工作仍在持续中。本节将介绍与行业发展状况和挑战问题相关的一些研究主题，同时也针对问题处理方式提出了自己的观点。

13.5.1 智能合约

该领域已经取得了重大进展，并制定了智能合约的关键需求和模板。然而，在智能合约的安全性方面，还需要开展进一步的研究工作。

13.5.2 中心化问题

在比特币采矿集中化方面，人们对比特币如何再次实现去中心化问题仍处于持续关注中。

13.5.3 加密功能的局限性

比特币区块链中的加密机制具有较强的安全性，且经受住了时间的考验。在其他区块链中，情况也大抵如此。然而，一些特定的安全问题仍处于持续研究中。例如，椭圆曲线的数字签名方案（导致私钥恢复攻击）、哈希函数中的碰撞问题，以及可能会破坏底层加密算法的量子攻击。

13.5.4 共识算法

权益证明算法或工作量证明的替代方法是一个重要的研究领域。这一认知观点主要源自下列事实：预计到2020年，比特币网络的电力消耗将达到140亿瓦。也有人认为，可抛弃某些效率低下或单一用途的工作，例如比特币的工作量证明算法；相反，网络力量可以用来解决一些数学或科学问题。另外，诸如权益证明的替代方案已经获得了众多

支持，并已在主区块链中实现，例如以太坊中的 Casper。然而，到目前为止，工作量证明算法仍然是保护公共区块链的最佳选择。

13.5.5 可扩展性

第 12 章已经对可扩展性进行了详细的讨论。简单地讲，虽然可扩展性问题已经取得了一些进展，但进一步的研究仍十分必要，进而实现区块链上的可扩展性，并改进区块链之外的解决方案，例如状态通道。相关方法包括：增加区块链尺寸和交易型区块链（不包含区块），从而提升区块链的自身功能，而不是使用侧通道，例如 IOTA（Tangle）。IOTA 定义为一个有向无环图（DAG），用于存储交易；而传统的交易区块链方案则采用区块存储交易，因而处理速度优于基于区块的区块链方案，例如比特币。其中，区块间的生成速度至少为 10 分钟。

13.5.6 代码混淆

第 12 章曾有所讨论，代码混淆（采用不可区分的混淆机制）可作为区块链中提供保密和隐私的一种手段。但该方案尚缺乏实际操作经验，同时也不是主流研究方向。

13.6 重要项目实例

本节列出了目前正在进行的区块链产业中值得注意的一些项目。除此之外，还有无数的初创企业和公司在从事区块链方面的工作，并发布了与区块链相关的产品。

13.6.1 以太坊上的 Zcash

以太坊上的 Zcash 则是 Ethereum R&D 团队近期发布的项目。其中，开发人员尝试使用 zk-SNARK（已在 Zcash 中加以实现）并针对以太坊创建一个隐私层。对于以太坊中的 Zcash，其目标在于创建一个平台并支持多项应用，例如隐私性较为重要的投票机制。另外，该平台还可创建以太坊上的匿名令牌，以供其他应用程序使用。

13.6.2 CollCo

CollCo 是由德意志银行开发的一个项目，该项目基于 Hyperledger 代码库，用于管理商业银行的现金转账。CollCo 提供了一个以区块链为基础的平台，可以实时转让商业银

行的资金，同时仍依赖于 Eurex Clearing CCP 提供的传统功能。该项目用于解决贸易结算过程中效率低下等问题。

13.6.3 Cello

截至 2017 年 2 月，Cello 是最新的 Hyperledger 项目。该项目旨在提供灵活的 BaaS 机制，进而方便用户实现多个区块链的部署和管理。据说，Cello 将支持所有目前和未来的 Hyperledger 区块链，如 Fabric 和 Sawtooth Lake。

13.6.4 Qtum

Qtum 项目源自比特币和以太坊的功能组合。Qtum 使用比特币代码库，但使用以太坊的 EVM 执行智能合约。Ethereum 智能合约可以使用比特币的 UTXO（未动用的交易）模型运行。

13.6.5 Bitcoin-NG

Bitcoin-NG 是另一个解决比特币区块链可扩展性、吞吐量和速度问题的协议。下一代 NG 协议基于领导者选举机制，一旦产生交易即对其进行验证。与比特币协议相比，在 Bitcoin-NG 协议中，区块间的时间以及区块尺寸是关系到可扩展性的关键限制条件。

13.6.6 Solidus

这是一种新的加密货币，它为自私的挖掘提供了解决方案，同时解决了伸缩性和性能问题。它还解决了保密问题。这是基于拜占庭式的共识。该协议目前的状态比较复杂，是一个开放的研究领域。

13.6.7 Hawk

该项目旨在解决区块链中智能合约的隐私问题。作为一个智能合约系统，支持区块链上的交易加密。Hawk 能够生成一个安全的协议，无须人工编程的加密协议就可以自动地与区块链进行交互。

13.6.8 Town-Crier

该项目旨在为智能合约提供真实的反馈结果，该系统基于英特尔的 SGX 可信硬件技

术。这在 Oracle 设计中更进了一步，智能合约可以在保护机密的同时请求来自在线资源的数据。

13.6.9 SETLCoin

SETLCoin 是一个由高盛集团开发的系统，并申请了证券结算应用程序加密货币专利。顾名思义，这种加密货币可以用于快速有效的结算。该技术利用虚拟钱包在网络对等点之间交换资产，并通过 SETLCoin 的所有权实现即时结算。

13.6.10 TEEChan

TEEChan 是使用可信执行环境的一种新方法，针对比特币区块链的扩展，提供了一种高效的解决方案。TEEChan 类似于支付通道的概念，即使用链外通道实现快速的交易传输。TEEChan 的关键之处在于，它在比特币区块链上是可实现的，而不需要对比特币网络进行任何改变，因为 TEEChan 是一个链外解决方案。然而，需要注意的是，由于英特尔 SGX CPU 用于提供 TEE，因此该解决方案需要使用到英特尔的远程认证（验证）。但是，这并非是非去中心化区块链中的理想属性。注意，由于远程认证（Intel）无法看到用户之间的通信内容，即使采用了远程认证机制，仍然会保持交易的机密性。在当前方案的去中心化和可信度方面，这种限制条件的使用仍值得商榷。

13.6.11 Falcon

Falcon 是一个扩展比特币规模的项目，提供了一个快速的中继网络，用于在网络上传播比特币。Falcon 的核心思想围绕着减少孤立区块展开，进而提高比特币网络的整体可扩展性。用于此目的的相关技术称作应用程序级剪通（cut-through）路由。

13.6.12 Bletchley

Bletchley 项目由微软公司推出，表明微软对区块链技术的承诺。Bletchley 允许使用 Azure 云服务并以用户友好的方式构建区块链。其中，Bletchley 引入的一个主要概念叫作 cryptlet，可认为是位于区块链之外的一个高级版本的 Oracle，并可通过使用安全通道的智能合约来调用。相关内容可以采用任何语言编写并在一个安全的容器中执行。

这里，存在两种类型的 cryptlet，即应用型 cryptlet 以及合约型 cryptlet。第一种类型提供了基本服务，例如加密和外部资源的数据抓取；而第二种类型则视为一种高级版本，并可在生成智能合约时自动创建，且位于区块链之外，同时保持着与链上合约间的

链接。由于位于区块链之外，因而没有必要在区块链网络的所有节点上执行合约型 cryptlet，因此该方法可提升区块链的性能。

13.6.13 Casper

Casper 是处于开发阶段的以太坊权益证明算法，预计将在 2017 年予以实施。这些节点在 Casper 以太坊网络中表示为可担保的验证器，并被要求支付保证金，以便生成新的区块。

13.6.14 Metropolis

Metropolis 是下一个版本的以太坊，其中大部分工作已处于完成阶段。当实现该版本时，需要使用到一个硬分叉。对此，一系列的 EIP 已被发布，其中包含了对该版本的所有重大改进。相应的改进措施包括将签名验证和 nonce 逻辑转移到智能合约中，对于安全方案的选取，这将为开发人员提供更灵活的机制。另一项改进措施则与区块哈希和状态根变化有关，进而可简化协议并支持并行交易处理。其他建议还包括对加密函数的大整数 (big integer) 支持 (椭圆曲线密码学)。上述所有的改进内容都包含在 EIP86、EIP98、EIP96、EIP100、EIP116、EIP116、EIP 19、EIP 140 和 EIP 141 上。目前，具体的发布日期尚不确定，同时也不清楚是否所有的 EIP 都将在新版本中实现。然而，当前版本的主要目的是为了简化协议，并提升隐私保护和灵活性。

13.7 其他工具

本节将简要介绍一些开发工具，以便让读者意识到区块链的各种开发选项，包括可用于区块链开发的平台和实用工具。

13.7.1 Microsoft Visual Studio 的 Solidity 扩展

该扩展提供了智能提示、自动完成和去中心化应用程序模板，并在熟悉的 Visual Studio IDE 环境下工作，以使研发人员更快地进入以太坊开发状态。

13.7.2 MetaMask

MetaMask 是一个 DAPP 浏览器，从 DAPP 浏览器视角来看与 Mist 十分类似，但允

许用户在浏览器中运行以太坊去中心化应用程序，且无须运行一个完整的以太坊节点。读者可访问 <https://metamask.io/> 下载 MetaMask，并作为 Chrome 插件进行安装。

13.7.3 Stratis

Stratis 是一个区块链开发平台，允许创建自定义的私有区块链，并出于安全原因与主 Stratis 区块链（Stratis 链）协同工作。Stratis 允许配置主区块链，如比特币、以太坊和 Lisk easy。此外，还支持使用 C#.Net 进行开发，也可以通过微软 Azure 作为 BaaS 予以提供。读者可访问 <https://stratisplatform.com/> 下载 Stratis。

13.7.4 Embark

Embark 是一个用于以太坊的开发框架，且与之前讨论的 Truffle 具有类似的功能。Embark 允许自动部署智能合约，并可方便地与 JavaScript 集成，特别是 IPFS。Embark 是一个功能丰富的框架，其中涵盖了多项功能，可通过 npm 进行安装。读者可访问 <https://github.com/iurimatias/embark-framework> 下载该框架。

13.7.5 DAPPLE

DAPPLE 是另一个用于以太坊的框架，通过处理更复杂的任务简化开发和部署智能合约。DAPPLE 可以用于包管理、合约构建和部署脚本，也是通过 npm 予以提供的。读者可访问 <https://github.com/nexusdev/dapple> 下载该框架。

13.7.6 Meteor

Meteor 是一个针对单页应用程序的全栈开发框架，可用于以太坊 DAPP 的开发。Meteor 中配置了相应的开发环境，从而简化 DAPP 的开发过程。读者可访问 <https://www.meteor.com/> 下载该框架；而与以太坊构建新信息相关的内容，则可访问 <https://github.com/ethereum/wiki/wiki/Dapp-using-Meteor>。

13.7.7 uPort

uPort 平台建立在以太坊之上，并提供了一个去中心化的身份管理系统。uPort 允许用户完全控制自己的身份和个人信息。uPort 基于信用机制，从而使用户能够彼此验证并建立信任。读者可访问 <https://www.uport.me/> 下载 uPort。

13.7.8 INFURA

该项目旨在提供企业级的以太坊和 IPFS 节点。INFURA 由以太坊节点、IPFS 节点和一个名为 Ferryman 的服务层组成，并提供路由和负载平衡服务。

13.8 与其他行业的结合

第 12 章曾讨论了区块链与物联网之间的融合。简单地说，由于区块链的真实性、完整性、隐私性和共享性质，物联网网络将会受益于区块链技术的发展。这可通过运行于区块链上的物联网网络加以实现，并利用去中心化的网格网络进行通信，以实现机器-机器（M2M）间的实时通信。由于 M2M 通信所产生的所有数据均可在机器学习过程中使用，故可以增强人工智能 DAO 或简单 AA 中的各项功能。这里，AA 可以作为区块链提供的分布式人工智能（DAI）环境中的智能体，并可以通过机器学习处理过程提升自身的智能。

人工智能是一门计算机学科并致力于构造智能体，根据环境制定理性的决策。机器学习在人工智能中起着至关重要的作用，并利用原始数据作为学习资源。在人工智能系统中，获取可靠的数据十分重要，这一类数据可用于机器学习和模型构建。从物联网设备、智能手机和其他数据采集中得到的大量数据，意味着人工智能和机器学习正变得越来越强大。然而，数据的真实性同样十分重要。一旦消费者、生产者和其他实体均位于区块链上，这些实体之间交互产生的真实数据就可以作为机器学习引擎的输入内容。可以认为，如果物联网设备被黑客入侵，则会将危险数据发送到区块链。由于物联网设备是区块链中的一个节点，并将所有安全属性应用于区块链网络中的一个正常节点上，因此这一问题得到了有效的缓解。相关属性包括对良好行为的激励、对危险交易的拒绝、严格的交易验证，以及区块链协议中的其他检查。因此，即使物联网设备被黑客入侵，也会被区块链网络视为一个拜占庭节点来对待，且不会对网络造成任何负面影响。

除此之外，智能 Oracle、智能合约和 AA 的结合将创建人工智能去中心化的自治组织（AIDAO），并可代表人类自身运行整个组织。这也体现了人工智能的另一个方面，并在未来可能成为常态。然而，为了实现这一愿望，还需要进行大量的研究工作。

此外，区块链技术与其他领域的融合还包括 3D 打印、虚拟现实、增强现实和游戏产业。例如，在多人在线游戏中，区块链的去中心化方案可以提高透明度，并确保不存在任何中央机构通过操纵游戏规则获得不公平的优势。所有这些话题都是目前较为活跃的研究领域，相信在不久的将来会得到更多的关注。

13.9 未来发展

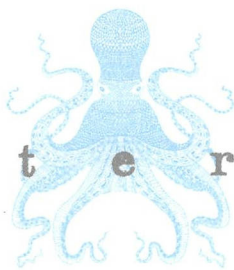
本节很少谈论区块链技术进展速度的详细预测，所有预测观点都可能在 2020—2050 年予以实现，如下所示：

- ❑ 物联网将在多个区块链上运行，并形成 M2M 经济链。
- ❑ 医疗记录将被安全地共享，同时保护医疗服务提供者私人区块链之间的病例隐私。这很可能是所有服务提供者之间共享的一个私有区块链。
- ❑ 选举机制将通过去中心化的 Web 应用程序进行，并通过透明和安全的方式使用区块链后端应用。
- ❑ 金融机构将运行大量的私人区块链，在参与者和内部流程之间共享数据。
- ❑ 金融机构将利用半私人区块链，为反洗钱行动和客户信息提供身份验证功能，并将在世界各地的金融机构之间共享。
- ❑ 移民和边境监管将记录在区块链上，护照监控将在世界各地的所有入境口岸和边境机构之间的区块链中实施。
- ❑ 密码学和分布式系统方面的研究将达到新的高度，大学和教育机构将开设与密码经济学、加密货币和区块链相关的专业课程。
- ❑ 人工智能 DAO（机器经济学）将出现于区块链上，并代表人类制定理性决策。
- ❑ 公共可监管的区块链将出现于日常生活中。
- ❑ 对于希望在区块链上运行业务或日常事务的人士，BaaS 将作为一项标准以供其使用。事实上，就像互联网一样，区块链将会与我们的日常生活无缝对接，人们在应用过程中无须了解其基础技术和基础设施。
- ❑ 区块链用于向艺术作品和媒体提供 DRM（数字版权管理）服务，同时向消费者提供有价值的内容，使消费者与生产者之间直接沟通，进而消除中心机构的许可和版权管理过程。
- ❑ 现有的加密货币（例如比特币）将处于持续增长状态，随着状态通道和可扩展性的增强，这种趋势只会不断增长。
- ❑ 加密货币投资将大幅增加，一个新的经济型社会将会出现。
- ❑ 比特币的价值将达到单位货币数万美元。
- ❑ 数字身份将在区块链上实现常规化管理，不同的政府职能，如投票、税收和资金支出，将通过区块链所支持的平台予以实施。
- ❑ 金融机构和结算机构将在 2017 年年底或 2018 年年初开始为客户推出基于区块链的解决方案。

13.10 本章小结

区块链将改变世界的格局，这种变化其级数将会呈现指数增长。本章探讨了区块链技术的各个项目和发展现状。首先讨论了相关技术的趋势，随着技术的进一步发展，这一态势也将不断升级。世界各地的许多研究人员和机构都对区块链技术予以极大的关注，本章也介绍了相关的研究课题。此外，本章还讨论了物联网和人工智能等其他领域间的融合。最后，还对区块链技术的发展进行了大胆的预测。大多数预测观点可能在未来 10 年左右实现，而有些内容可能需要更长的时间方得以实现。区块链技术有可能改变世界，一些积极的迹象体现在：某些概念证明已成功地实现。同时，越来越多的技术狂热者和开发者对这项技术产生了浓厚的兴趣。很快，区块链将与我们的生活交织在一起，就像现在的互联网一样。本章只是对区块链的巨大潜力进行了简要的介绍，相信在不久的将来，对这一技术的运用量将呈指数级增长。

M a s t e r i n g



本书全面介绍了区块链技术的理论和实践，涵盖了理解区块链技术的全部内容。在阅读完本书后，读者将能够深入了解区块链技术的内部工作原理，并具备开发区块链应用程序的能力。本书包含了与区块链技术相关的所有主题，涉及密码学、加密货币、比特币、以太坊等，以及用于区块链开发的各种平台和工具。

— 本书主要包含以下特点：

- 掌握区块链技术的理论和技术基础。
- 深入理解去中心化的概念、影响及其与区块链技术的关系。
- 通过具体实例，讲解如何使用加密技术来保护数据。
- 掌握区块链内部的工作原理，以及比特币等加密货币背后的相关机制。
- 探讨智能合约的理论基础。
- 区块链技术在其他领域的应用。
- 介绍其他区块链解决方案，包括Hyperledger，Corda等。
- 探索区块链技术的研究主题和未来发展趋势。

B l o c k c h a i n

清华社官方微信信号



扫 我 有 惊 喜

ISBN 978-7-302-49983-1



9 787302 499831 >

定价：129.00元